Spring 5-7-2012

Conformational Switching and Graph Grammar: Abstract Models of Self-Assembly

Arnab Dutta

CONFORMATIONAL SWITCHING AND GRAPH GRAMMAR: ABSTRACT MODELS OF SELF-ASSEMBLY

by

ARNAB DUTTA

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
Department of Mathematics

Casey Mann, Ph.D., Committee Chair
Regan Beckham, Ph.D., Committee Chair

College of Arts and Sciences

The University of Texas at Tyler
May 2012

The University of Texas at Tyler
Tyler, Texas

This is to certify that the Master's Thesis of

ARNAB DUTTA

has been approved for the thesis requirement on
April 17, 2012
for the Masters of Mathematics degree

Approvals:

Thesis Chair: Regan Beckham, Ph.D.

Thesis Chair: Casey Mann, Ph.D.

Member: Nathan Smith, Ph.D.

Chair, Department of Mathematics

Dean, College of Arts and Science

Table of Contents

List Of Figures

Abstract

CONFORMATIONAL SWITCHING AND GRAPH GRAMMAR :
ABSTRACT MODELS OF SELF-ASSEMBLY

Arnab Dutta

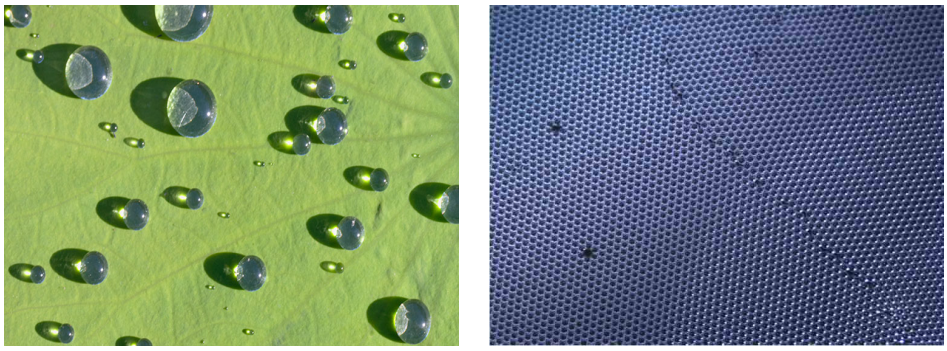
Thesis Chairs: Casey Mann, Ph.D.
Regan Beckham, Ph.D.

The University of Texas at Tyler
May 2012

Abstract models of self-assembly are mathematical models that capture the behavior of natural or artificial self-assembling systems. In 1995, conformational switching in self-assembling mechanical systems was introduced and an abstract model of self-assembling systems, conformational switching model, was later developed where assembly instructions are written as rules representing the conformational changes of self-assembling components. In 2004, another abstract model named graph grammar was developed to self-assemble a prespecified graph structure by generating a grammar or a set of rules. We first provide the concepts related to the abstract models of self-assembly, followed by a brief history of the development of conformational switching and graph grammar approaches. An overview of the conformational switching model is provided, including descriptions of two types of conformational switches and the theory of one-dimensional self-assembling automata. A description of the graph grammar model is also provided, including its topological properties and an algorithm for generating graph grammar.

1. Introduction

In a very simple and straightforward definition, *self-assembly* refers to the process, natural or artificial, in which objects autonomously come together to form larger complexes. There are numerous examples of self-assembly in nature. One very simple example is water droplets on a lotus leaf, coming together spontaneously to form a larger drop as depicted in Figure 1(a). Consider *bubble rafts* where small or large soap bubbles form a larger pattern of soap bubbles as in Figure 1(b). Crystalline structures such as the salt crystal shown in Figure 2 or the structure of a *tobacco mosaic virus* are results of self-assembly.



(a) Water drops on a lotus leaf (b) Bubble Raft. Source: [14]

Figure 1: (a) Water drops on a lotus leaf self-assemble together to form a big drop (b) Example of a bubble raft

Indeed self-assembly has application in many fields. Engineers are building conformational switches which has applications in microelectromechanical (MEMS) and nanoelectromechanical (NEMS) devices. Biologists are trying to solve the *protein folding problem* which concerns how the basic subunits or amino acid sequences of a protein determine its folded structure. Chemists are studying the processes of forming larger and stable molecular structures polymerization. Computer scientists are working on DNA computers to make computation faster. Mathematicians are developing theoretical abstract models to capture and control the

Figure 2: Salt(NaCl) crystal. Source: [15]

behavior of physical self-assembling systems. Our goal in this thesis is to understand these efficient abstract mathematical models that represent physical self-assembling systems. We begin by introducing abstract models of self-assembly. We conclude the introduction with a brief timeline of historical developments related to the discovery and advancements of abstract models of self-assembly. Throughout this section, we refer to [4], [12] and Pelesko's textbook on self-assembly titled "Self-Assembly: The Science of Things That Put Themselves Together"[10].

1.1. Abstract Models of Self Assembly

Most of the works on self-assembly are based on experiments or physical models. We often find similarities among different physical self-assembling systems. For example, in the self-assembly of a *tobacco mosaic virus*, self assembled protein disks switch to a lockwasher configuration while interacting with RNA, changing its conformation [2, 3]. Similarly, proteins always change its conformation while interacting with other proteins. In order to capture similar behavior in self-assembling systems, we need an abstract or general model. These abstract models may help us find the answers to questions like "what type of assembly or structure is possible?"by achieving *coded self-assembly* where the self-assembling components of the system carry information about the final structure. Also one major

advantage of these abstract models is they are computationally solvable; that is, we can write computer algorithms to simulate the behavior of these models rather than running time consuming and costly experiments [10]. In the next sections we will discuss two types of abstract models : the *conformational switching model* and the *graph grammar model*.

1.2. History and Background

Historically the idea of conformational switching in self-assembling systems was first introduced by Kazuhiro Saitou and Mark J. Jackiela [6]. In their 1995 paper [6] Saitou and Jackiela designed self-assembling mechanical systems where assembly instructions can be written in terms of conformational switches. They showed that mechanical components capable of changing conformations can be used to encode any particular subassembly sequence and nondesirable sequences can be blocked or minimized, therefore maximizing the yield of desired subassembly sequences. In 1996, Saitou and Jackiela published another paper titled [7], in which they construct a self-assembling automaton (a sequential rule-based machine) capable of capturing the functions of conformation switches. In short, their abstract model of self-assembly answered questions like "Is it possible to encode any particular subassembly sequence?" or "How many conformations are necessary to encode the given subassembly sequence?"

In 2004, Eric Klavins and his collaborators introduced a new approach of self-assembly called *graph grammar* [4]. They developed an experimental self-assembling robotic system consisting of *programmable parts* or robots and showed how we can use graph grammar or assembly rules to control the self-assembly of the programmable parts. They consider the conformation of a part as a symbol and the final assembly structure as a simple graph labeled by such symbols. The initial system is a simple graph where all the vertices or robotic components are labeled by the same symbol without any edge between them and the graph grammar consists

of assembly rules that are pairs of simple labeled graphs. If a subset of vertices or self assembling components together with their labels and edges matches the left hand side graph of some rule, then the subset can be replaced by the right hand side graph of the rule. If we continue to apply rules we may be able to produce several stable assemblies or an unique stable assembly depending on the type of rule set or grammar.

There are other abstract models of self assembly besides conformational switching and graph grammar. One very interesting model is *Tile Assembly Model* developed by Erik Winfree and his collaborators [13, 11, 9]. Their work was motivated by the natural self assembly of DNA helices. They showed that the tile assembly model is *Turing Universal* by self-assembling a Sierpinski Triangle using two-dimensional self-assembly of DNA tiles which are similar to the *Wang Tiles* developed by Hao Wang in 1961. Their work provides an useful link between self-assembly and algorithmic computation.

In this thesis, we focus particularly on the two abstract models : *the conformational switching model* and *the graph grammar model*.

2. Conformational Switching

In this section, we discuss our first abstract model, *conformational switching model*, developed by Kazuhiro Saitou and Mark J. Jackiela[12, 6, 7]. Many biological structures are results of self-assembly where components change their conformations while interacting with other components. A good example is *tobacco mosaic virus* or TMV. Figure 3 shows electron microgrph of TMV particles. It has rod-like appearance, few hundred nanometers in length and about 18 nanometers in diameter. These rod-like structures are actually helical and these helices are made of proteins and RNA. In the self-assembly of *tobacco mosaic virus*[10, 3, 2] proteins assemble themselves into "washer" or disk-like configuration, as shown in Figure 4.

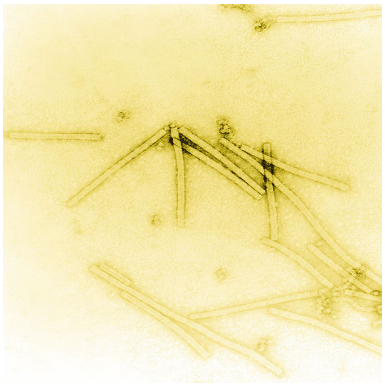


Figure 3: Electron micrograph image of TMV particles. Source:[17]

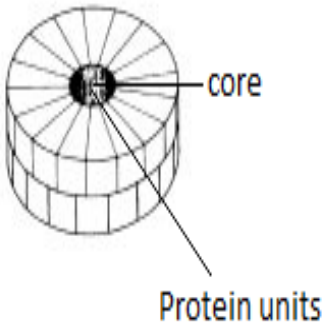
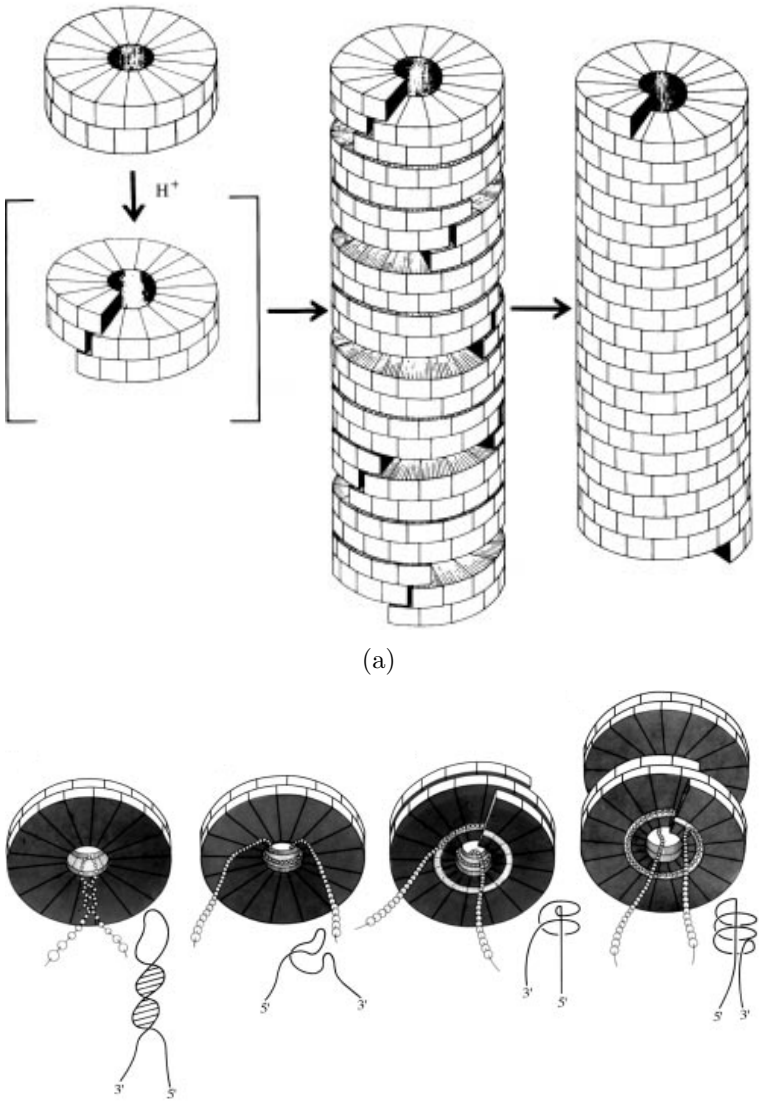


Figure 4: Protein disk in the assembly of TMV

A hairpin loop of RNA enters into the core of the protein disk or washer configuration and as a result, the washer changes its conformation and takes a different configuration called "lockwasher" configuration, as illustrated in Figure 5. As a result, a portion of the RNA gets trapped inside the protein disks. In this new configuration, another protein disk can bind to the first one by changing its conformation and by repetition the rod-like structure of tobacco mosaic virus is assembled. Biologists believe that the assembly instructions for this kind of conformational change are written in the protein units or components as *conformational switches*.



(a)

(b)

Figure 5: Conformational change in the assembly of tobacco mosaic virus. Source:[2, 3]

Saitou and Jackiela came up with a novel approach to build self-assembling mechanical systems where assembly instructions can be written in the mechanical parts or components as conformational switches [12, 6, 7]. They also provided an abstract model, *a one-dimensional self-assembling automaton*, where assembly instructions are represented as rules imitating the behavior of conformational switches.

To understand how conformational switches encode an assembly sequence, consider an example scenario, as illustrated in [10] by Pelesko. Consider a system

consisting of a mixture of three types of particles A, B and C. These particles are square tiles which have certain bond sites on its edges. These particles are pictured in 6 .

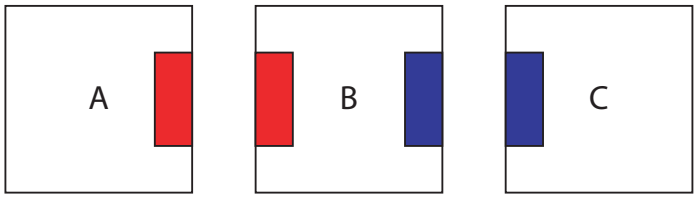


Figure 6: Particles with different bond sites

In Figure 6 particle A has one bond site on its right edge colored red, particle B has two bond sites on its left and right edges colored red and blue respectively and particle C has one bond site on its left edge colored blue so that A can bond to B on the left side and C can bond to B on the right side. Also A, B and C cannot bond to themselves, they can only bond to each other. Assume we have a large container containing equal numbers of A, B and C particles. At each step we take two random particles or clusters out of the container and bind them if they have the same colored edges. Then we return the particles, or cluster to the container and repeat the process. If we continue this process we will be left with a container of ABC clusters. But there is more than one way we can construct ABC clusters. Here we have two possible subassembly sequences. These bonding or sequences can be seen as *rules* as follows

$$\begin{cases} A + B \rightarrow AB \\ B + C \rightarrow BC \\ AB + C \rightarrow ABC \\ A + BC \rightarrow ABC \end{cases} \tag{1}$$

Equation (1) indicates two approaches which we can follow to form ABC clusters – we can either form a BC and then add an A or we can form AB and then add a C. Saitou and Jackiela [12, 7] introduced a tree notation to represent the assembly

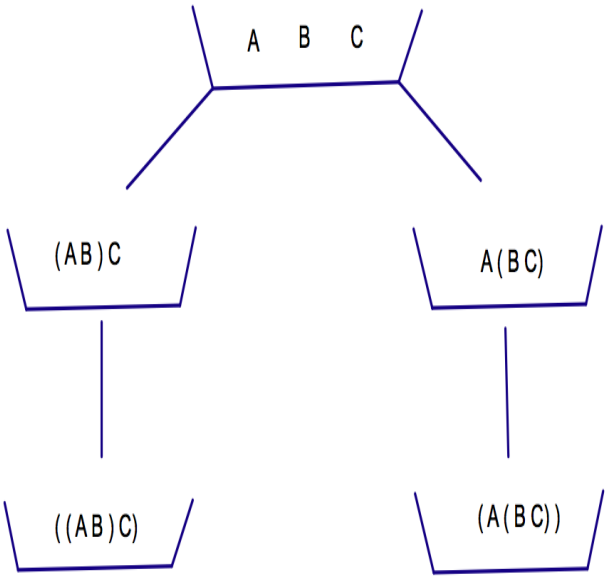


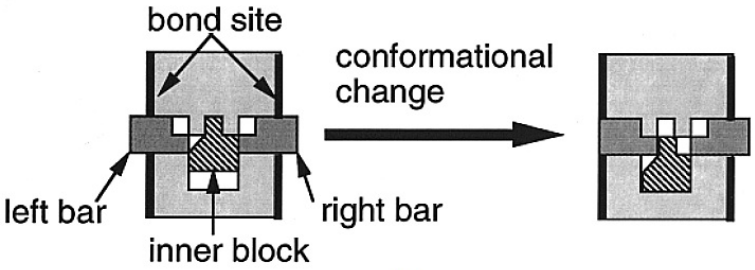
Figure 7: Self-assembly with no conformational switches: ABC can assemble in two ways. Source: [10]

of ABC clusters. To keep track of different subassembly sequences we parenthesize every time a cluster is formed, indicating the order of the assembly. So in this case the subassembly sequences would look like $(A(BC))$ and $((AB)C)$ depending on the order of assembly. The tree notation for this assembly is depicted in Figure 7.

In the previous example, the rules do not enforce any subassembly sequences to assemble ABC, i.e. the final container may contain either $((AB)C)$ or $(A(BC))$ or both depending on the order in which the rules are being *fired*. According to Saitou and Jackiela [12, 6, 7] one can enforce a particular subassembly sequence by designing *conformational switches*, which are like assembly instructions built in the components. A conformational switch causes the formation of a bond at one site and may be used to change the conformation of another site, thus blocking or enabling other subassembly sequences. In the next subsection we give a detailed description of a special type of conformational switch called *Minus Device* and how minus device improves the *yield* of the self-assembling system. Again throughout the next subsections we refer to the work of Saitou and Jackiela [12, 6, 7] and the book [10] of Pelesko.

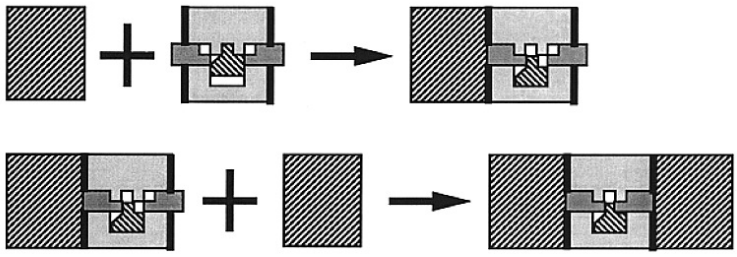
2.1. Minus Devices

In the previous example, there are two equally likely subassembly seuqences. Saitou and Jackiela [12, 6, 7] introduced the idea of a conformational switch which enforces a particular subassembly sequence. These switches are built-in mechanisms inside the mechanical components or particles. Saitou and Jackiela called these switches *minus devices*.



(a) mechanism

(b) notation



(c) interaction with another particle

Figure 8: A mechanical conformational switch: Minus Device (a) mechanism (b) notation (c) interaction with another particle. Source: [12]

Figure 8(a) shows a patricle with a minus device in it. It has right and left bars that can slide horizontally, and one interior sliding block that can slide vertically. In Figure 8(a) we also see the status of the minus device before and after the conformational change. Before conformational change the left bar can be pushed in. But the right bar can not be pushed in due to the interior block position.

When some other particle or component pushes the left bar in, the interior block slides down so that the right bar can be pushed in. Figure 8(b) shows a notational diagram of a minus device. The arrow indicates the direction in which the horizontal bars can be pushed in. Figure 8(c) shows an example assembly of two particles or components where one of the two particles or components has a minus device in it and the other one has no switching mechanism. Since only the left bar can be pushed in before the particle changes its conformation, the particle with no switching mechanism must come from the left to bond. Once the left bar is pushed in, both of these particles or components bond together and the right bar becomes free to be pushed in. Then another particle with no switching mechanism comes from the right, and pushes the right bar and attached itself to the cluster. Now since the right bar cannot be pushed in before a particle or component comes from the left and pushes the left bar, we see a left to right assembly.

Now we go back to our previous example, with no conformational switching, and build a minus device inside the particle B. With our previous assumption, C particles can only bind to the B particles on the right, which is possible only when some A particle comes from the left side and pushes the left bar. This causes the right bar to automatically open up. Now when an A particle from the left pushes the left bar, B particle changes its conformation and makes the right bar free to be pushed in. We can denote the new conformation of B particle as $B^{'}$. Then a C particle comes from the right and pushes the right bar and attach itself to the $(AB^{'})$ cluster, resulting in an assembly $((AB^{'})C)$. Recall that the parentheses denote the order in which the assembly takes place. We can view this assembly in terms of rules of the form,

$$\begin{cases} A + B \rightarrow AB^{'} \\ B^{'} + C \rightarrow B^{'}C \end{cases} \tag{2}$$

We see these rules enforce only one subassembly sequence, denoted $((AB)C)$. So

conformational switches allow us to enforce a particular subassembly sequence. As Pelesko [10] has noted, though the above system is simple enough to understand how the minus devices encode a particular subassembly sequence, it does not make it easy to see the effect of conformational switching or minus devices on the *yield* of a self-assembling system. One example scenario, as given by Pelesko[10], is a system of A, B and C particles as shown in Figure 9(a) where a B particle can bind to an A particle first and then a C particle can bind to the AB cluster, forming a complete square ABC cluster, as shown in Figure 9(b) or a C particle can bind to an A particle first, forming an AC cluster only shown in Figure 9(c). Our goal is to form ABC clusters. So the possible assemblies are as follows,

$$\begin{cases} A + B \rightarrow AB \\ AB + C \rightarrow ABC \\ A + C \rightarrow AC \end{cases} \tag{3}$$

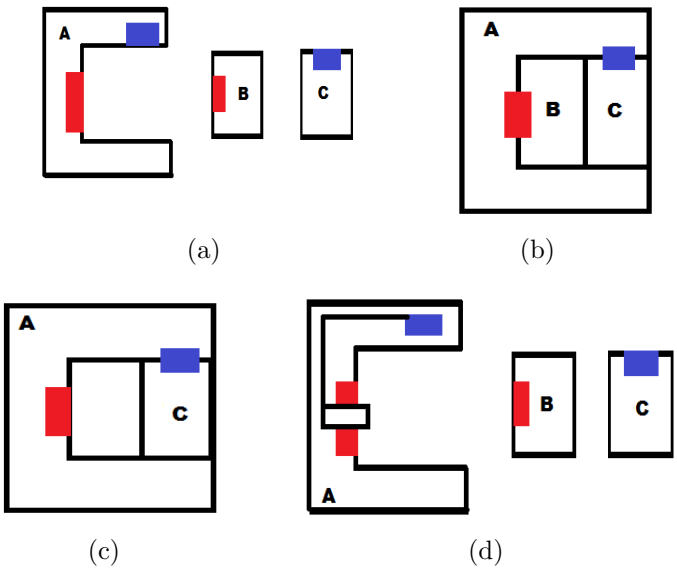


Figure 9: Improving the yield of a system: (a) no conformational switch (b) ABC complex (c) AC complex (d) System with conformational switch

Certainly the yield of this system, the output of ABC clusters, may not be very high since once an AC cluster is formed, a B particle can not attach itself to AC. Now suppose we have a system like the one shown in Figure 9(d) where A particle

has a conformational switch in it. In this case, a B particle must bind to A first, pushing the bar, following which a C particle binds to the AB cluster, forming a complete square ABC cluster as shown in Figure 9(b). The possible assemblies in this case are given below.

$$\begin{cases} A + B \rightarrow A^{'}B \\ A^{'}B + C \rightarrow A^{'}BC \end{cases} \tag{4}$$

Thus conformational switching increases the yield of the system by blocking the formation of AC clusters.

2.2. Assembly of Four Particles and Plus Devices

So far we have seen the application of conformational switching in a three particle system. Imagine a system consisting of a mixture of four particles A, B, C and D. Our goal is to assemble an ABCD complex. As Saitou and Jackiela noted, five non-ambiguous subassembly sequences are possible: $(((AB)C)D)$, $((AB)(CD))$, $((A(BC))D)$, $(A((BC)D))$ and $(A(B(CD)))$. They also noted that three of these five non-ambiguous subassembly sequences, given $(((AB)C)D)$, $((AB)(CD))$, $(A(B(CD)))$, are encodable by minus devices and remaining two subassembly sequences, given $((A(BC))D)$ and $(A((BC)D))$, can not be encoded by minus devices due to the fact that propagation of conformational change through the particles can not be done by minus devices. By simulating the above system using a genetic algorithm, Saitou and Jackiela noticed that the two unencodable subassembly sequences $((A(BC))D)$ and $(A((BC)D))$ yield better than other best encodable sequences like $(A(B(CD)))$ or $(A(B(CD)))$. To encode these sequences Saitou and Jackiela introduced another type of comformational switch called *plus device* which has a sliding bar that allows propagation of conformational change through the particles or components.

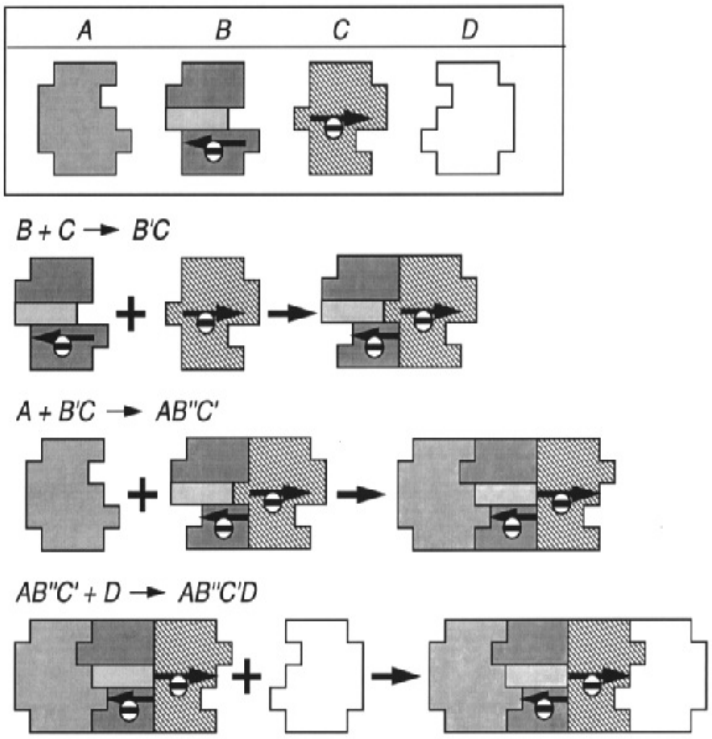


Figure 10: Conformational switching mechanism that encodes subassembly sequence $((A(BC))D)$. Source:[12]

Figure 10 illustrates the encoding of subassembly sequence $((A(BC))D)$ by plus and minus devices. Note that particle B has both a plus and a minus devices. Notice when particle A attaches itself to the cluster $B^{'}C$ a conformational change of particle B from $B^{'}$ to $B^{''}$ propagates to particle C resulting in a conformational change of particle C. As a result D particle can attach itself to the C particle and form an ABCD complex. Other subassembly sequences can also be encoded by some combination of plus and minus devices.

2.3. The Abstract Model: One-dimensional Self-assembling Automata

In the previous subsections, we have discussed how minus and plus devices can encode subassembly sequences, and increase the yield of a self-assembling system. But can we generalize this theory for any subassembly sequence? By generalization Saitou and Jackiela mean the following two questions:

- Is it possible to encode a given assembly sequence by using minus devices only, or by using minus and plus devices?
- If an assembly sequence is encodable, then how many conformations are necessary to encode the assembly sequence?

According to Saitou and Jackiela, the relationship between assembly sequences and conformational switches is analogous to the relation between languages and machines, where an assembly sequence is an instance of a language and a set of conformational switches which encode the assembly sequence is a machine that accepts the instance of the language.

To answer the previous questions, Saitou and Jackiela defined a formal model, called a *one-dimensional self-assembling automaton*. A one-dimensional self assembling automaton is a sequential rule-based machine that processes one-dimensio nal strings of symbols. Here the *rules* abstracts conformational changes and assembly instruction of symbols or components, where each component can take a finite number of conformations required for assembly.

Throughout the following subsections we refer to the work of Saitou and Jackiela [12, 7].

2.3.1. Terminology

In the following definitions and examples, a *component* is an element of a finite set Σ, and an *assembly* is a string in Σ^+. A *component* $a \in \Sigma$ can take a finite number of *conformations* denoted by $a, a', a'', a''' \ldots$, and the transition from one conformation to another is represented by a set of *assembly rules*, which represent the type of conformational switches, or more specifically the function of conformational switches.

Definition 1. A *one-dimensional self-assembling automaton*, abbreviated SA, is a pair $M = (\Sigma, R)$, where Σ is a finite set of *components*, and R is a finite set of

assembly rules of the form either $a^\alpha + b^\beta \rightarrow a^\gamma b^\delta$ or $a^\alpha b^\beta \rightarrow a^\gamma b^\delta$, where $a, b \in \Sigma$ and $\alpha, \beta, \gamma, \delta \in \{'\}^*$. indicating different conformations of the components, and also $a^\varepsilon = a$ where ε represents the null string.

Here the *conformation set* of a component $a \in \Sigma$ is a set $C_a = \{a^\alpha | \alpha \in \{'\}^*, a^\alpha$ appears in $R\}$. Hence the *conformation set* of SA, M, is the union of all conformation sets of all the components $a \in \Sigma$. The *rules* of the form $a^\alpha + b^\beta \rightarrow a^\gamma b^\delta$ are called *attaching rules* that represents the minus devices and the rules of the form $a^\alpha b^\beta \rightarrow a^\gamma b^\delta$ are called *propagation rules* that represents the plus devices.

Using the definition above , the self-assembling system in our previous three particle example can be defined as $M = (\Sigma, R)$, where $\Sigma = \{A, B, C\}$, and $R = \{A + B \rightarrow AB', B' + C \rightarrow B'C\}$. The conformation set of M is $C = \{A, B, B', C\}$.

According to Saitou and Jackiela in our previous example of the three particles A, B, C system SA M has a *component container* where there are infinite number of slots capable of storing an assembly or the null string ε. Suppose initially there are a finite number of slots containing assemblies and also a number of empty slots containing ε. Components are self-assembled by picking a random pair of assemblies or an assembly in the container and then applying the rules in R to the assemblies. After the application of each rule the resulting assemblies are deleted from and added to the container slots. According to Saitou and Jackiela one of following three outcomes is possible everytime one or two assemblies are picked:

1. If the two assemblies (x, y) are of the form $(za^\alpha, b^\beta u)$ for some $a, b \in \Sigma$, $z, u \in \Sigma^*$, and if R has a rule r of the form $a^\alpha + b^\beta \rightarrow a^\gamma b^\delta$ and r fires, delete x and y, and add $za^\gamma b^\delta u$.

2. If an assembly $x = za^\alpha b^\beta u$ for some $a, b \in \Sigma$, $z, u \in \Sigma^*$, and if R has a rule r of the form $a^\alpha b^\beta \rightarrow a^\gamma b^\delta$ and r fires, delete x and add $za^\gamma b^\delta u$.

3. If neither of the above applies, the pair of assemblies or one assembly are returned to the container, leaving the container unchanged.

So far we have talked about representing a self-assembling system in terms of components and rules. But we need to be able to describe the state of a self-assembling system at any point in the process of self-assembly. So we need a representation of the container listing present assemblies and we also have to keep a track of the sequence in which these assemblies have been formed.

Saitou and Jackiela defined *SEQ(A)* as the language generated by the context-free grammar $\forall a \in A,\ S \rightarrow (SS)|a$, where A is a finite set. Also note that $A \subset$ *SEQ(A)*. A string x in *SEQ(A)* is a full parenthesization of a string $u =$ RM-PAREN(x) in A^+, where RM-PAREN is a function that removes parentheses from its argument string.

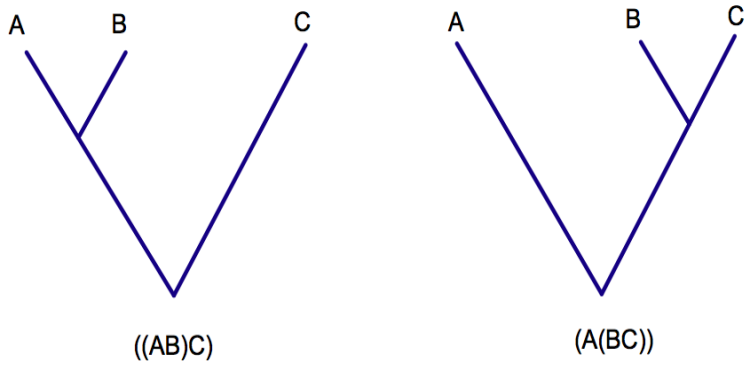


Figure 11: Binary tree representations of subassembly sequences

As we discussed before, we use the tree notation to represent an assembly sequence. Figure 11 illustrates binary tree representations of subassembly sequences $((AB)C)$ and $(A(BC))$ from the three particle system.

Definition 2. A *subassembly sequence* is a string in $SEQ(\Sigma)$. A subassembly sequence x is *basic* if x contains at most one copy of elements in Σ, that is, $\forall a \in \Sigma, N_a \leq 1$.

Let $M = (\Sigma, R)$ be an SA where $\Sigma = \{a, b, c\}$. Then the subassembly sequence $((ab)c)$ is *basic* since it contains only one copy of each component in Σ. Throughout this subsection we'll deal with basic sequences only.

Definition 3. Let $M = (\Sigma, R)$ be an SA. A *configuration* of M is a *bag* $\langle x | x \in SEQ(C) \rangle$, where C is the *conformation set* of M.

Let $x \in SEQ(\Sigma)$ be a subassembly sequence. A configuration Γ *covers* x if $\Gamma = \langle a | a \in \Sigma \rangle$ and $\forall a \in \Sigma, N_a(x) \leq NUM_a(\Gamma)$, where $NUM_a(\Gamma)$ is the number of as in Γ.

We can trace the sequence of any self-assembly process by carefully observing the configuration every time the component container changes due to the application of a rule from the rule set. We use the notation $\Gamma \vdash_M \Phi$ to denote the change of configuration of SA M from Γ to Φ as a result of applying a rule in the rule set R to the component container exactly once. Similarly, we use the notation $\Gamma \vdash^*_M \Phi$ to denote the change of configuration of M from Γ to Φ as a result of applying one or more rules in the rule set R to the component container zero or more times. The following example illustrates these concept more clearly.

In our previous three-particle system $M = (\Sigma, R)$, where $\Sigma = \{A, B, C\}$, and $R = \{A + B \rightarrow AB^{'}, B^{'} + C \rightarrow B^{'}C\}$. Let the configurations $\Gamma = \langle A, B, B, C, C \rangle$ and $\Phi = \langle A, B, C \rangle$. Now by Definition 3 both of these configurations cover the subassembly sequence $x = ((AB)C)$. To self-assemble x from Γ we apply rule $A + B \rightarrow AB^{'}$ first and then apply rule $B^{'} + C \rightarrow B^{'}C$.

$$\langle A, B, B, C, C \rangle \vdash_M \langle (AB^{'}), B, C, C \rangle \vdash_M \langle ((AB^{'})C), B, C \rangle \tag{5}$$

Self-assembly of any subassembly sequence terminates only when no rule firing is possible, otherwise it keeps running due to infinite times of rule firing. So it is pretty obvious to say that a self-assembling automaton assembles a particular string or sequence if the process of self-assembly terminates, and the terminating configurations must contain the string that is assembled in the sequence. Saitou and Jackiela called these configurations as *stable configurations*. The following definition gives a more formal and mathematical description of stable configurations.

Definition 4. Let $M = (\Sigma, R)$ be an SA, Γ be a configuration of M, and $x \in$

$SEQ(\Sigma)$ be a subassembly sequence. Γ is *stable* if there is no rule firing from Γ, that is, $S_M(\Gamma) = \{\Gamma\}$, where $S_M(\Gamma) = \{\Phi | \Gamma \vdash_M^* \Phi\}$

M terminates from Γ if all configurations derived from Γ can derive a stable configuration, that is, $\forall \Phi \in S_M(\Gamma), \exists \Phi_1 \in S_M(\Phi)$, such that $S_M(\Phi_1) = \{\Phi_1\}$. M *self-assembles* x from Γ if both of the following conditions hold:

1. M terminates from Γ.
2. $\forall \Phi \in S_M^*(\Gamma), \exists y \in \Phi$ such that $x =$ RM-PRIME(y), where $S_M^*(\Gamma)$ is a set of stable configurations derived from Γ, and RM-PRIME is a function that removes the prime ($'$) symbols from its argument.

Thus in the above three-particle system, M self-assembles $x = ((AB)C)$ from the configuration $\Gamma = \langle A, B, B, C, C \rangle$.

2.3.2. An Algorithm for Constructing One-dimensional Self-assembling Automata

In this section we consider the problem: Given a basic subassembly sequence $x \in SEQ(\Sigma)$, is it possible to write an algorithm to construct a set of rules R such that an SA $M = (\Sigma, R)$ self-assembles x from any configuration that covers x? According to Saitou and Jackiela [7], such an algorithm can be written since x can be represented as a parse tree that is a binary assembly tree. A computer implementation of this algorithm would definitely help us to derive the rules needed to self-assemble any assembly sequence without doing any time-consuming physical experiment. In Algorithm 1 [7] *GenerateRules* takes as input a basic subassembly sequence $x \in SEQ(\Sigma)$, a variable $flag \in \{left, right, none\}$ which indicates the direction of next assembly, and a set of rules or assembly instructions R. *GenerateRules*$(x, none, \emptyset)$ returns a pair (u, R), where u is the final assembly(with all conformations) such that RM-PRIME(u) =RM-PAREN(x) and R is the set of rules containing the assembly rules to assemble x from Γ. Also note that *INC* is

a function that increments the conformation of each component by adding a ($'$) symbol to its argument string . *LEFT* and *RIGHT* functions return the left and right end symbol of their argument strings.

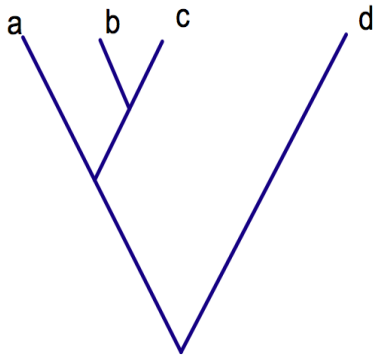


Figure 12: Parse tree representation of subassembly sequence $((a(bc))d)$

Let us illustrate Algorithm 1 with an example: let $\Sigma = \{a, b, c, d\}$ be a component set, and $x = ((a(bc))d)$ be a subassembly sequence. A configuration $\Gamma = \langle a, b, b, c, c, d\rangle$ covers x. A parse tree of x is shown in Figure 12. Since x is not a single component, line 1-2 is skipped. Now *GenerateRules* recursively traverses the left and right subtrees in line 3, starting with $y = (a(bc))$, $z = d$ and comes down to $y = b$, $z = c$, that is, at this time, *GenerateRules*$(b, right, R_1)$ and *GenerateRules*$(c, left, R_2)$ are called, where R_2 is the rule set returned by *GenerateRules*$(b, right, R_1)$. As *GenerateRules*$(b, right, R_1)$ returns (b, R_2) where $R_2 = \emptyset$ by line 1-2 and as a result *GenerateRules*$(c, left, R_2)$ returns (c, R_3) where $R_3 = R_2 = \emptyset$ by line 1-2. Now in function *GenerateRules*$((bc), left, R_3)$ $flag = left$, by line 7 $a^{\alpha} = b$, by line 8 $b^{\beta} = c$, and thus by line 13 a new attaching rule is added to the empty rule set R_3 and a new rule set R_4 is generated, where $R_4 = \{b + c \rightarrow b'c\}$ by line 14. Then in line 15 *PropagateLeft*(b, R_4) is called and returns (b', R_4) (by line 1-2 in the definition of *PropagateLeft*). Line 16 returns with $(b'c, R_4)$. Now in function *GenerateRules*$((a(bc)), right, R_4)$, $flag = right$, by line 7 $a^{\alpha} = a$, by line 8 $b^{\beta} = b'$, and thus by line 18 a new attaching rule of the form $a + b' \rightarrow ab''$ is added to the rule set R_4 by line 19. Then in line 20 *PropagateRight*$(b'c, R_4)$ is called and a propagation rule of the

Algorithm 1 *GenerateRules*$(x, flag, R)$

Require: x is a basic subassembly sequence

if $x = a$ **then**
  return (a, R)
else
  if $x = (yz)$ **then**
    $(u, R) \leftarrow GenerateRules(y, right, R)$
    $(v, R) \leftarrow GenerateRules(z, left, R)$
    $a^{\alpha} \leftarrow RIGHT(u)$
    $b^{\beta} \leftarrow LEFT(v)$
    if $flag = none$ **then**
      $R \leftarrow R \cup \{a^{\alpha} + b^{\beta} \rightarrow a^{\alpha}b^{\beta}\}$
      return (uv, R)
    end if
    if $flag = left$ **then**
      $R \leftarrow R \cup \{a^{\alpha} + b^{\beta} \rightarrow a^{INC(\alpha)}b^{\beta}\}$
      $(u, R) \leftarrow PropagateLeft(u, R)$
      return (uv, R)
    end if
    if $flag = right$ **then**
      $R \leftarrow R \cup \{a^{\alpha} + b^{\beta} \rightarrow a^{\alpha}b^{INC(\beta)}\}$
      $(v, R) \leftarrow PropagateRight(v, R)$
      return (uv, R)
    end if
  end if
end if

FUNCTION DEFINITION: $PropagateLeft(u, R)$

if $u = a^{\alpha}$ **then**
  return $(a^{INC(\alpha)}, R)$
end if
if $u = va^{\alpha}b^{\beta}$ **then**
  $R \leftarrow R \cup \{a^{\alpha}b^{INC(\beta)} \rightarrow a^{INC(\alpha)}b^{INC(\beta)}\}$
  $(u, R) \leftarrow PropagateLeft(va^{\alpha}, R)$
  return $(ub^{INC(\beta)}, R)$
end if

FUNCTION DEFINITION: $PropagateRight(u, R)$

if $u = a^{\alpha}$ **then**
  return $(a^{INC(\alpha)}, R)$
end if
if $u = a^{\alpha}b^{\beta}v$ **then**
  $R \leftarrow R \cup \{a^{INC(\alpha)}b^{\beta} \rightarrow a^{INC(\alpha)}b^{INC(\beta)}\}$
  $(u, R) \leftarrow PropagateRight(va^{\alpha}, R)$
  return $(a^{INC(\alpha)}u, R)$
end if

form $b''c \rightarrow b''c'$ is added to the rule set R_4 resulting in a new rule set, say $R_5 = \{b + c \rightarrow b'c, a + b' \rightarrow ab'', b''c \rightarrow b''c'\}$ (by line 4-6) and $(b''c', R_5)$ is returned. Now line 21 returns with $(ab''c', R_5)$. Next we go back to the function $GenerateRules(((a(bc))d), none, R_5)$ where $flag = none$, by line 7 $a^{\alpha} = c'$, by line 8 $b^{\beta} = d$. Thus by line 9 a new attaching rule of the form $c' + d \rightarrow c'd$ is added and a new rule set, say $R_6 = \{b+c \rightarrow b'c, a+b' \rightarrow ab'', b''c \rightarrow b''c', c' + d \rightarrow c'd\}$ is generated in line 10. Line 11 returns with $(ab''c'd, R_6)$. Also note that RM-PRIME $(ab''c'd)$ =RM-PAREN$((a(bc))d)$.

Thus an SA $M = (\Sigma, R)$, discussed above, self-assembles x from any configuration that covers x.

2.3.3. Classes of One-dimensional Self-assembling Automata

As we have discussed before, there are two types of rules in a one-dimensional self-assembling automaton - *attaching rules* and *propagation rules*. Attaching rules abstract the function of minus devices, whereas propagation rules abstract the function of plus devices. Two classes of self-assembling automata can be defined based on these two different rule types.

Definition 5. Let $M = (\Sigma, R)$ be an SA. We call M *class I* if R contains only attaching rules. M is *class II* if R contains both attaching and propagation rules.

We can also categorize basic subassembly sequences based on these two different types of classes of SA. The basic subassembly sequences which correspond to the class I SA are those in which the direction from which new components are added does not change during the entire self-assembly process. But in case of basic subassembly sequences that correspond to the class II SA the direction of self-assembly must change at least once. The following discussion gives a more detailed description of these classes and their corresponding subassembly sequences.

Definition 6. Let $M = (\Sigma, R)$ be an SA. An *assembly template* is a string $t \in SEQ(\{p\})$. An *instance* of t on Σ is a subassembly sequence $x \in SEQ(\Sigma)$

obtained by replacing each p in t by some $a \in \Sigma$. If x is an instance of t, then t is an *assembly template* of x.

Let two strings $t_1 = (((pp)p)(pp))$ and $t_2 = (((p(pp))p)p)$ be assembly templates, and $\Sigma = \{a, b, c, d, e\}$. Then the basic subassembly sequences $x_1 = (((ab)c)(de))$ and $x_2 = (((b(ac))e)d)$ are instances of t_1 and t_2 on Σ, respectively.

Definition 7. An *assembly grammar* is a context-free grammar with a language that is a subset of $SEQ(\{p\})$. The class I assembly grammar G_I can be defined by the following substitution rules:

$$G_I = \begin{cases} S \to (LR) \\ L \to (Lp)|p \\ R \to (pR)|p \end{cases} \tag{6}$$

Here S is called the *start variable*. To generate a string or sequence, first we write down the start variable. Then we replace the variable with right hand side of the rule corresponding to that variable and repeat the process until no variables (S, L, R) remain. The *language* of G_I, denoted $L(G_I)$, is the set of all strings or sequences that can be generated by the context-free grammar G_I. The assembly template $t_1 = (((pp)p)(pp))$ can be generated by G_I, through the following derivation:

$$\begin{cases} S \to (LR) \\ \quad \to (L(pR)) \\ \quad \to (L(pp)) \\ \quad \to ((Lp)(pp)) \\ \quad \to (((Lp)p)(pp)) \\ \quad \to (((pp)p)(pp)) \end{cases} \tag{7}$$

and hence $t_1 \in L(G_I)$.

Likewise the assembly template $t_3 = ((((pp)p)p)(p(pp)))$ can also be generated by G_I:

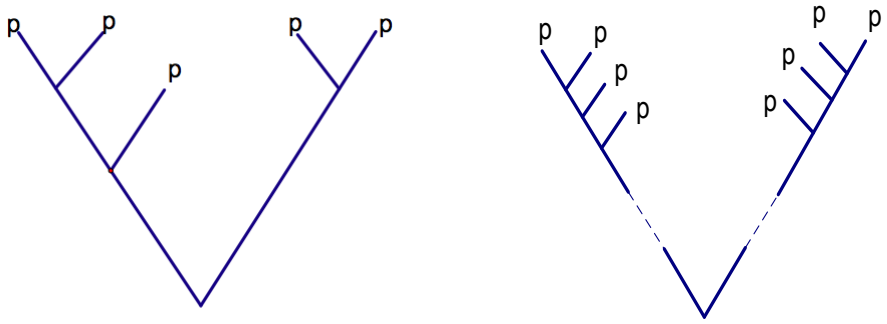
$$\begin{cases} S \to (LR) \\ \quad \to ((Lp)R) \\ \quad \to (((Lp)p)R) \\ \quad \to (((Lp)p)(pR)) \\ \quad \to (((Lp)p)(p(pR))) \\ \quad \to ((((Lp)p)p)(p(pR))) \\ \quad \to ((((pp)p)p)(p(pR))) \\ \quad \to ((((pp)p)p)(p(pp))) \end{cases} \tag{8}$$

The structure of assembly templates in $L(G_I)$ can be generalized by

$$(((\cdots((pp)p)\cdots)p)(p(\cdots(p(pp))\cdots))) \tag{9}$$

Figure 13 illustrates the parse tree or general binary tree of the assembly templates in $L(G_I)$. Each of the left and right subtrees is a linear assembly tree, indicating that the direction of assembly does not change during the self-assembly process.

The class II assembly grammar G_{II} can also be defined by the following substitution rules:



(a) (b)

Figure 13: (a) Parse tree of the assembly template $t_1 = (((pp)p)(pp))$, (b) Generalized parse tree of an assembly template generated by G_I. Source: [7]

$$G_{II} = \begin{cases} S \to (L_0 R_0) \\ L_0 \to (L_0 R_1) | R_1 \\ R_0 \to (L_1 R_0) | L_1 \\ L_1 \to (L_1 p) | p \\ R_1 \to (p R_1) | p \end{cases} \tag{10}$$

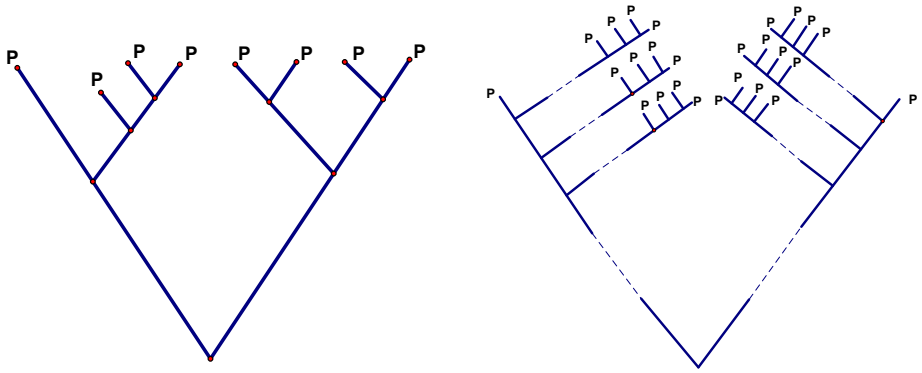
The *language* of G_{II}, denoted by $L(G_{II})$, is the set of all strings or sequences that can be generated by the context-free grammar G_{II}. The assembly template $t_4 = ((p(p(pp)))((pp)(pp)))$ can be generated by G_{II}, through the following derivation:

$$
\begin{cases}
S \to (L_0R_0) \\
\quad \to ((L_0R_1)R_0) \\
\quad \to ((L_0(pR_1))R_0) \\
\quad \to ((L_0(pR_1))(L_1R_0)) \\
\quad \to ((L_0(p(pR_1)))((L_1p)R_0)) \\
\quad \to ((L_0(p(pR_1)))((L_1p)(L_1R_0))) \\
\quad \to ((R_1(p(pR_1)))((L_1p)(L_1R_0))) \\
\quad \to ((R_1(p(pR_1)))((L_1p)(L_1L_1))) \\
\quad \to ((p(p(pR_1)))((L_1p)(L_1L_1))) \\
\quad \to ((p(p(pR_1)))((L_1p)(pL_1))) \\
\quad \to ((p(p(pp)))((L_1p)(pL_1))) \\
\quad \to ((p(p(pp)))((pp)(pL_1))) \\
\quad \to ((p(p(pp)))((pp)(pp)))
\end{cases}
\tag{11}
$$

Figure 14 illustrates the parse tree or general binary tree of the assembly templates in $L(G_{II})$. Note that the direction of the assembly changes during the self-assembly process.

From the above discussion we can see that for any assembly template $t \in L(G_I)$, the direction of self-assembly does not change during the self-assembly of any particular instance x of t. But if t belongs to the complement $SEQ(\{p\}) \setminus L(G_I)$, then the direction of self-assembly changes at least once during the self-assembly of x, instance of t. The following theorems are some important results based on these observations.

Theorem 1. *For any basic subassembly sequence x that is an instance of an assembly template $t \in L(G_I)$, there exists a class I SA which self-assembles x from a configuration that covers x.*

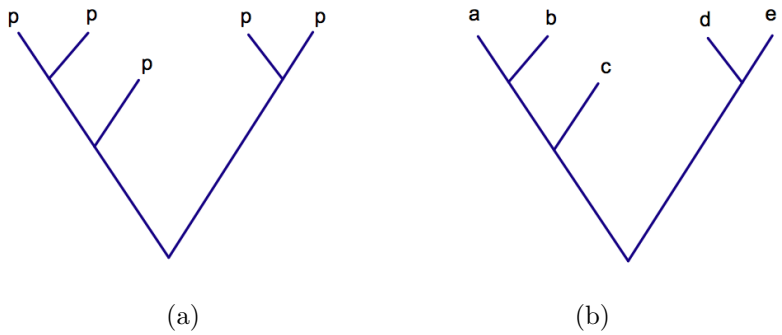


(a) (b)

Figure 14: (a) Parse tree of the assembly template $t_4 = ((p(p(pp)))((pp)(pp)))$, (b) Generalized parse tree of an assembly template generated by G_{II}. Source: [7]

The above statement is an answer to the first of the two questions, as discussed in section 1.3, which were the main focus of the work of Saitou and Jackiela. This important result proves that a basic subassembly sequence in $L(G_I)$ can be generated by using minus devices or attaching rules only. The proof of this theorem follows Algorithm 1, since it suffices to show that the rule set returned by $GenerateRules(x, none, \emptyset)$ contains no propagation rules. Please refer to [7] for a detailed and formal proof of Theorem 1. Here we give an outline of the original proof by demonstrating an example assembly of a basic subassembly sequence $x = (((ab)c)(de))$ which is an instance of the previously mentioned assembly template $t_1 = (((pp)p)(pp))$ which belongs to $L(G_I)$. Figure 15(a) shows a parse tree of t_1.

Since the depth of the parse tree $D(t) = 3$, we can write $t = (l_{d_l} r_{d_r})$ where $d_l = 2$ is the depth of left subtree, $d_r = 1$ is the depth of the right subtree, $l_k = (l_{k-1}p)$ for $k = 1, 2$ and $r_k = (pr_{k-1})$ for $k = 1$, and $l_0 = r_0 = p$. Let y_k and z_j be substrings of x of l_k and r_j, respectively. Now if we refer back to Algorithm 1, $GenerateRules(x, none, \emptyset)$ recursively calls $GenerateRules(y_{d_l}, right, \emptyset)$ and $GenerateRules(z_{d_r}, left, R_1)$ in lines 5-6 , where $y_{d_l} = ((ab)c)$, $z_{d_r} = (de)$, and R_1 is the rule set returned by $GenerateRules(y_{d_l}, right, \emptyset)$. Let R_2 be the rule set



(a) (b)

Figure 15: (a) Parse tree of the assembly template $t_1 = (((pp)p)(pp))$, (b) Parse tree of an assembly sequence $x = (((ab)c)(de)$

returned by $GenerateRules(z_{d_r}, left, R_1)$. According to Algorithm 1 propagation rules are added only after $GenerateRules(x, none, \emptyset)$ returns in line 11. We have to show that both R_1 and R_2 do not contain any propagation rule. By induction it can be easily shown that R_1 does not contain any propagation rule, since $GenerateRules(y_{d_l}, right, \emptyset)$ recursively calls $GenerateRules((ab), right, \emptyset)$ and $GenerateRules(c, left, \tilde{R}_1)$, where by induction hypothesis $\tilde{R}_1$ does not contain any propagation rule and $GenerateRules(c, left, \tilde{R}_1)$ returns $\tilde{R}_1$ by line 2 in Algorithm 1. Now the condition in line 16 is satisfied and an attaching rule is added to $\tilde{R}_1$ in line 17, resulting in a new rule set $\hat{R}_1$. After that, in line 18 $PropagateRight(c, \hat{R}_1)$ is called, returning the final rule set $R_1 = \hat{R}_1$, which has no propagation rule. Using the same mathematical induction on z_{d_r}, we can also prove that R_2 does not contain any propagation rule.

Theorem 2. *For any basic subassembly sequence x that is an instance of an assembly template $t \in SEQ(\{p\}) \setminus L(G_I)$, there exists a class II SA which self-assembles x from a configuration that covers x.*

The significance of Theorem 2 is that it proves that any basic subassembly sequence which is not a member of $L(G_I)$ can be generated by a combination of plus and minus devices. Once again the reader is asked to refer to [7] for an original and detailed proof of the above theorem by Saitou and Jackiela. Here is a brief outline of the proof with an example. The proof largely follows Algorithm 1, since it suffices to show that the rule set returned by $GenerateRules(x, none, \emptyset)$ contains

atleast one propagation rule. Suppose we have a basic subassembly sequence $x = ((a(bc))d)$ which is an instance of the assembly template $t_5 = ((p(pp))p) \in SEQ(\{p\})$. Figure 12 illustrates the parse tree of x.

It can be easily shown that $t_5 \notin L(G_I)$, that is, it can not be generated by using minus devices or attaching rules only. Since the depth of the parse tree of t_5, $D(t_5) = 3$, we can write $t_5 = (l^l_{d_l} r^r_{d_r})$, where $d_l = 2$ is the depth of the left subtree, $d_r = 1$ is the depth of right subtree, $l^l_i = (l^l_{i-1} l^r_{i-1})$ for $i = 1, \ldots, d_l$ and $r^r_i = (r^l_{i-1} r^r_{i-1})$ for $i = 1, \ldots, d_r$, and $l^l_0 = r^r_0 = p$. Since $t_2 \notin L(G_I)$ there exists a $j \in \{1, \ldots, d_l\}$, in this case 2, such that, the length $L(l^r_j) = 2$ (and/or there exists a $j \in \{1, \ldots, d_r\}$, in this case none, such that the length $L(r^l_j) = 2$). When $j = 2$, $L(l^r_j) = 2$. So $y^l_j = a$, $y^r_j = (bc)$ and $y^l_{j+1} = (a(bc))$ are substrings of x corresponding to l^l_j, l^r_j, and l^l_{j+1}, respectively. Let R_0 be the rule set containing no propagation rule. We have to show that the rule set returned by $GenerateRules(y^l_{j+1}, right, R_0)$ contains at least one propagation rule. Now since $y^l_{j+1} = (a(bc))$, $GenerateRules(y^l_{j+1}, right, R_0)$ recursively calls $GenerateRules(a, right, R_0)$ and $GenerateRules((bc), left, R_1)$ in lines 5-6 of Algorithm 1, where R_1 is the rule set returned by $GenerateRules(a, right, R_0)$. After $GenerateRules((bc), left, R_1)$ returns, the condition in line 18 is satisfied and attaching rule $(a + b^{'} \rightarrow ab^{''})$ is added to R_2, line 19. We call this new rule set $\hat{R_2}$. Then in line 20 $PropagateRight(ab^{''}c, \hat{R_2})$ is called and the condition in line 4 in the subroutine is satisfied and at this point a propagation rule $(b^{''}c \rightarrow b^{''}c^{'})$ is added to $\hat{R_2}$. Therefore the rule set returned by $GenerateRules(y^l_{j+1}, right, R_0)$ contains a propagation rule.

Since the function of a propagation rule or plus device cannot be replaced by attaching rules or minus devices, it is not possible to self-assemble a class II basic subassembly sequence stably and reliably by using a class I SA.

2.3.4. Minimum Conformations in Self-assembly

In the previous section, we have discussed the problem of whether a given subassembly sequence is encodable or not using minus and plus devices. In this section, we focus on the problem: how many conformations are necessary to encode a given subassembly sequence? Our goal is to find the minimum number of conformations necessary to self-assemble a given basic subassembly sequence stably and reliably. As we have noted in earlier examples, in a self-assembly process,the number of conformations of each component are not same. Therefore, we focus on the maximum number of conformations of all components.

Definition 8. Let M be an self-assembling automaton(SA). The number of *conformations* n of M is defined by,

$$n = max\{\alpha | a^{\alpha} \in C\} \tag{12}$$

where C is the *conformation set* of M as defined earlier.

From our earlier discussion, there are three categories of assembly templates- 1) assembly templates in $L(G_I)$, the language associated with class I grammar, 2) assembly templates in $L(G_{II})$, the language associated with class II grammar, and 3) assembly templates in the set $SEQ(\{p\})\backslash L(G_{II})$, that is, templates that are not in the languages of class I and II grammar. The minimum number of conformations that is necessary for the self-assembly of a basic subassembly sequence x depends on the category of template it belongs to.

Theorem 3. *For any basic subassembly sequence x that is an instance of an assembly template $t \in L(G_I)$, there exists a class I SA with two conformations which self-assembles x from a configuration that covers x. If the length of x, $L(x) \geq 3$, M is an SA with the minimum number of conformations which self-assembles x from a configuartion that covers x.*

According to Theorem 3, for any basic subassembly sequence x that is an in-

stance of assembly templates in $L(G_I)$ the minimum number of conformations required to self-assemble x is two. This statement makes sense since any attaching rule generated by *GenerateRules* requires maximum two conformations for each component. For a detailed proof of this theorem refer to [7]. For a basic subassembly sequence that is an instance of an assembly template in $L(G_{II}) \setminus L(G_I)$, Saitou and Jackiela stated the following.

Theorem 4. *For any basic subassembly sequence x that is an instance of an assembly template $t \in L(G_{II}) \setminus L(G_I)$, there exists a class II SA M with two conformations which self-assembles x from a configuration that covers x. And M is an SA with the minimum number of conformations which self-assembles x from a configuartion that covers x.*

Consider our previous example where $\Sigma = \{a, b, c, d\}$ and $x = ((a(bc)d)$ which is an instance of the assembly template $t = ((p(pp))p)$ in $L(G_{II}) \setminus L(G_I)$. $GenerateRules(x, none, \emptyset)$ returns with $(ab''c'd, R)$ where R is the rule set containing $\{b + c \rightarrow b'c, a + b' \rightarrow ab'', b''c \rightarrow b''c', c' + d \rightarrow c'd\}$. We can see that $M = \{\Sigma, R\}$ is a class II SA with two conformations which self-assembles x from a configuration that covers x.

For basic subassembly sequences x that are instances of assembly templates in $SEQ(p) \setminus L(G_{II})$ Saitou and Jackiela showed that only three conformations are necessary to self-assemble x from any configuration that covers x. A detailed discussion on these results can be found in [7].

3. Graph Grammar

In this section, we discuss another abstract model, *graph grammar model*, developed by Eric Klavins and his collaborators [1, 4, 5]. Klavins et al. designed triangular programmable robotic tiles and showed how graph grammar can be used to direct self-assembly of these robotic tiles. They modeled an assembly as a simple graph labeled by discrete symbols that represent the conformation of the components. Vertices of the graph represent the self-assembling components, and an edge between two components represents that they are attached. Most importantly assembly rules are pair of labeled graph rather than reaction mechanism between two components as shown in the conformational switching model. Though graph grammar model is somehow motivated by conformational switching model of Saitou and Jackiela, it focuses on the topological structure of assembly rather than geometrical or physical structures of assemblies or components.

Throughout this section and following subsections, we refer to Klavins et al. work [4, 5].

3.1. Terminology

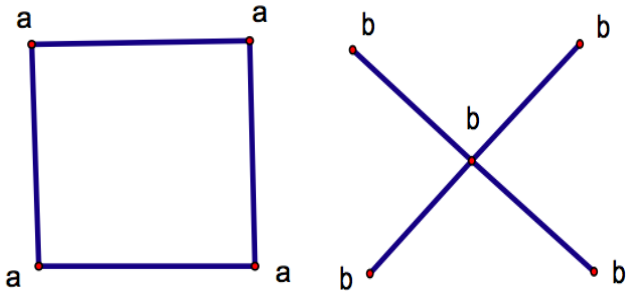
In this section we provide definitions and examples of basic concepts related to graph grammar.

Definition 9. A *simple labeled graph* over an alphabet Σ is a triple $G = (V, E, l)$ where V is a set of vertices, E is a set of pairs of vertices from V, and $l : V \to \Sigma$ is a labeling function which maps the vertices in V to the alphabet, Σ(i.e. the labeling function l gives each vertex a name).

We denote an edge $\{x, y\} \in E$ by xy. By graph Klavins et. al means a *network topology* of an interconnected collection of robots where label $l(x)$ of robot x corresponds to the *state* of the robot and keeps track of the local information. Figure 13 illustrates two example graphs. The graph in Figure 16(a) can be de-

noted as $G_1 = (\{1,2,3,4\}, \{\{1,2\},\{2,3\},\{3,4\},\{4,1\}\}, \lambda x.a)$ using lambda calculus notation and the graph in (b) is denoted as

$G_2 = (\{1,2,3,4,5\}, \{\{1,2\},\{1,3\},\{1,4\},\{1,5\}\}, \lambda x.b)$ using the same notation.



(a) (b)

Figure 16: Examples of simple labeled graph

Definition 10. For two graphs G_1 and G_2 , $f : G_1 \to G_2$ or $f : V_{G_1} \to V_{G_2}$ means that f is a function from the vertex set of G_1 to the vertex set of G_2. A function $h : G_1 \to G_2$ is a *label preserving embedding* or *label preserving monomorphism* if

- h is one-to-one,
- $\{x, y\} \in E_{G_1}$ if and only if $\{h(x), h(y)\} \in E_{G_2}$.
- $l_{G_1} = l_{G_2} \circ h$.

If h is onto then it is called an *isomorphism*.

Definition 11. A *rule* is a pair of graphs $r = (L, R)$ where $V_L = V_R$ and the graphs L and R are called the *left hand side* and *right hand side* of r respectively.

The *size* of rule r is $|V_L| = |V_R|$. A *rule* is named unary, binary or ternary depending on the number of vertices in V_L. A set of rules is called a *rule set* or *grammar*. Here is an example of a rule set or grammar Φ which contains binary and ternary rules

$$\Phi = \begin{cases} a \quad a \Rightarrow b - c, & (r_1) \\ a \quad d \Rightarrow c - e, & (r_2) \\ b \diagup^{d} \diagdown e \;\Rightarrow\; b' \diagup^{d} \diagdown e\,, & (r_3) \end{cases} \tag{13}$$

We represent a binary rule graphically as $a \quad a \Rightarrow b - c$. In this rule, $V_L = V_R = \{1, 2\}$ and vertex 1 is labeled as $l_L(1) = a$ in the left hand side and as $l_R(1) = b$ in the right hand side. Note that $a \quad a \Rightarrow b - c$ is a binary rule since $|V_L| = |V_R| = 2$. Similarly, $b \diagup^{d} \diagdown e \;\Rightarrow\; b' \diagup^{d} \diagdown e$ is a ternary rule since $|V_L| = |V_R| = 3$.

Definition 12. A rule r is *applicable* to a graph G if there exists a label preserving embedding $h : V_L \to V_G$, that is, We call this function h a *witness*.

That is, we find a similarity between the left hand side graph L of a rule and the graph G. An *action* on a graph G is a pair (r, h) such that r is applicable to G with witness h. Applying an action (r, h) on a graph $G = (V, E, l)$ returns a new graph $G' = (V', E', l')$ defined by

$$V' = V$$

$$E' = (E - \{\{h(x), h(y)\} | \{x, y\} \in E_L\}) \cup \{\{h(x), h(y)\} | \{x, y\} \in E_R\}$$

$$l'(x) = \begin{cases} l(x) & \text{if} \quad x \notin h(V_L), \\ l_R \circ h^{-1}(x) & \text{otherwise.} \end{cases}$$

We write $G \xrightarrow{r,h} G'$ to denote that G' is obtained from G by the application of (r, h). For example, let us consider the graph G_2 in Figure 13(b) and assume we have a rule r of the form $b \quad b \Rightarrow c - c$ in the rule set Φ . It is easy to find a label preserving embedding or witness between the left hand side graph of the rule r and the graph G_2. Therefore, we can apply an action (r, h) on the graph G_2. Figure 17 shows an application of rule r on the graph G_2.

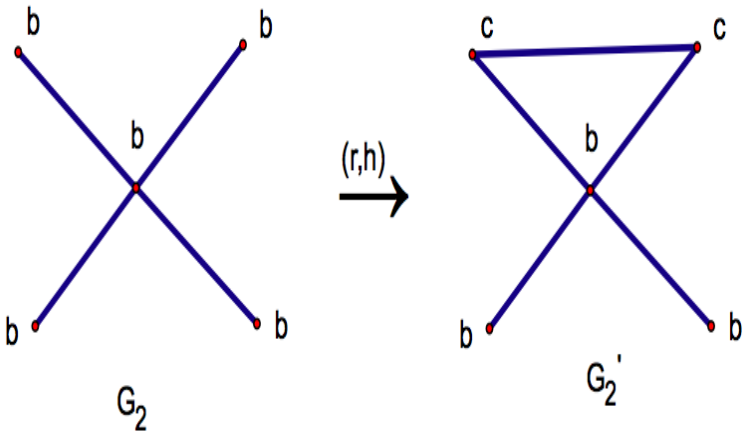


Figure 17: An application of rule r on the graph G_2

After application of (r, h), the resultant graph $G_2^{'}$ has the following structure:

$$V_2^{'} = V_2 = \{1, 2, 3, 4, 5\}$$

$$E_2^{'} = \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{1, 5\}, \{2, 3\}\}$$

As we defined before the labeling function $l^{'}(x)$ for the new resultant graph $G_2^{'}$ is same as the labeling function $l(x)$ of the graph G_2 except for the vertices 2 and 3 that are replaced by the vertices 1 nad 2 of right hand side graph R of the rule r. Also note that the set of edges for the new graph $G_2^{'}$ has one extra edge $\{2, 3\}$ since right hand graph R of the rule r has an edge between the vertices $\{1, 2\}$ in R and since $h(1) = 2$ and $h(2) = 3$ in graph G_2.

Definition 13. A *graph assembly system* is a pair (G_0, Φ) where G_0 is the *initial graph* of the system and Φ is a set of rules. Klavins et. al identify a system by its rule set or grammar Φ and assume that the initial graph is a infinite graph defined by

$$G_0 \equiv (N, \emptyset, \lambda x.a)$$

Where $a \in \sum$ is the *initial symbol*.Here $\lambda x.a$ is the *lambda calculus* notation

for the function assigning the label a to all vertices. That is, the initial graph G_0 looks like the one given in Figure 18, where all the vertices are labeled a.

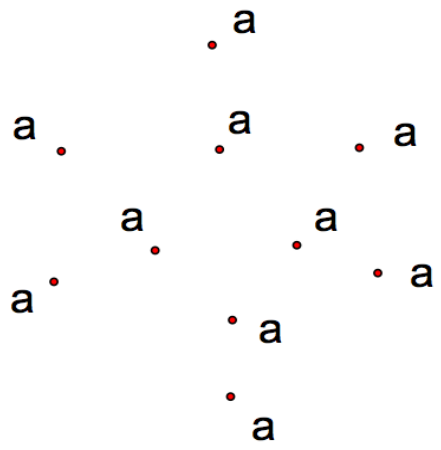


Figure 18: An example initial graph

Definition 14. A *trajectory* or *assembly sequence* of a system (G_0, Φ) is a (finite or infinite) sequence

$$G_0 \xrightarrow{r_1,h_1} G_1 \xrightarrow{r_2,h_2} G_2 \xrightarrow{r_3,h_3} \dots$$

According to Klavins et al. , in a finite sequence there is no rule in Φ applicable to the terminal graph. A system can have more than one trajectory since we are dealing with nondeterministic systems. We denote the set of trajectories of a system by $\tau(G_0, \Phi)$.

Definition 15. A graph G is *reachable* in the system (G_0, Φ) if there exists a trajectory $\sigma \in \tau(G_0, \Phi)$ with $G = \sigma_k$ for some k. The set of all such reachable graphs is denoted by $\mathcal{R}(G_0, \Phi)$.

Klavins et al. were particularly interested in the connected components of reachable graphs, as these correspond to the collection of robots that are connected by physical links.

Definition 16. A connected graph H is a *reachable component* of a system (G_0, Φ) if there exists a graph $G \in R(G_0, \Phi)$ such that H is a component of G. The set of all such reachable components is denoted by $\mathcal{C}(G_0, \Phi)$. A reachable component may be temporary. That is there may be some rule in Φ that operates

on part of it.

Definition 17. A component $H \in \mathcal{C}(G_0, \Phi)$ is *stable* if, whenever H is a component of $G_k \in R(G_0, \Phi)$ via a monomorphism f, then H is also a component via f of every graph in $\mathcal{R}(G_k, \Phi)$. The stable components are denoted by $S(G_0, \Phi) \subseteq C(G_0, \Phi)$.

That is, the stable components are those that no applicable rule can change.

We illustrate all the definitions in this subsection by describing a self-assembly system that assembles chains and cycles from individual parts, as demonstrated by Klavins et al. [4, 5]. First of all, we need a rule set Φ. Suppose our rule set Φ is defined by

$$\Phi = \begin{cases} a \quad a \Rightarrow b - b, \quad (r_1) \\ a \quad b \Rightarrow b - c, \quad (r_2) \\ b \quad b \Rightarrow c - c, \quad (r_3). \end{cases}$$

Here the first rule r_1 in Φ is given by the pair of graphs $L = (\{1, 2\}, \emptyset, \lambda x.a)$ and $R = (\{1, 2\}, \{\{1, 2\}\}, \lambda x.b)$, where L and R are left hand side and right hand side of the rule r_1 respectively. The initial graph G_0 in this case is given by

$$G_0 \equiv (N, \emptyset, \lambda x.a)$$

since all the robots are initially labeled by the symbol a. Now we have an initial graph G_0 in Figure 19. At first, the only rule we can apply is r_1 since all of the vertices in G_0 is labeled by the symbol a. In order to apply a rule r_1 from the rule set Φ, we have to find a label-preserving embedding h from the vertex set of the left hand side graph of r_1 to the vertex set of the initial graph G_0. It can easily be shown that there exists such an embedding since both of these vertex sets have vertices labeled by the symbol a only and their corresponding sets of

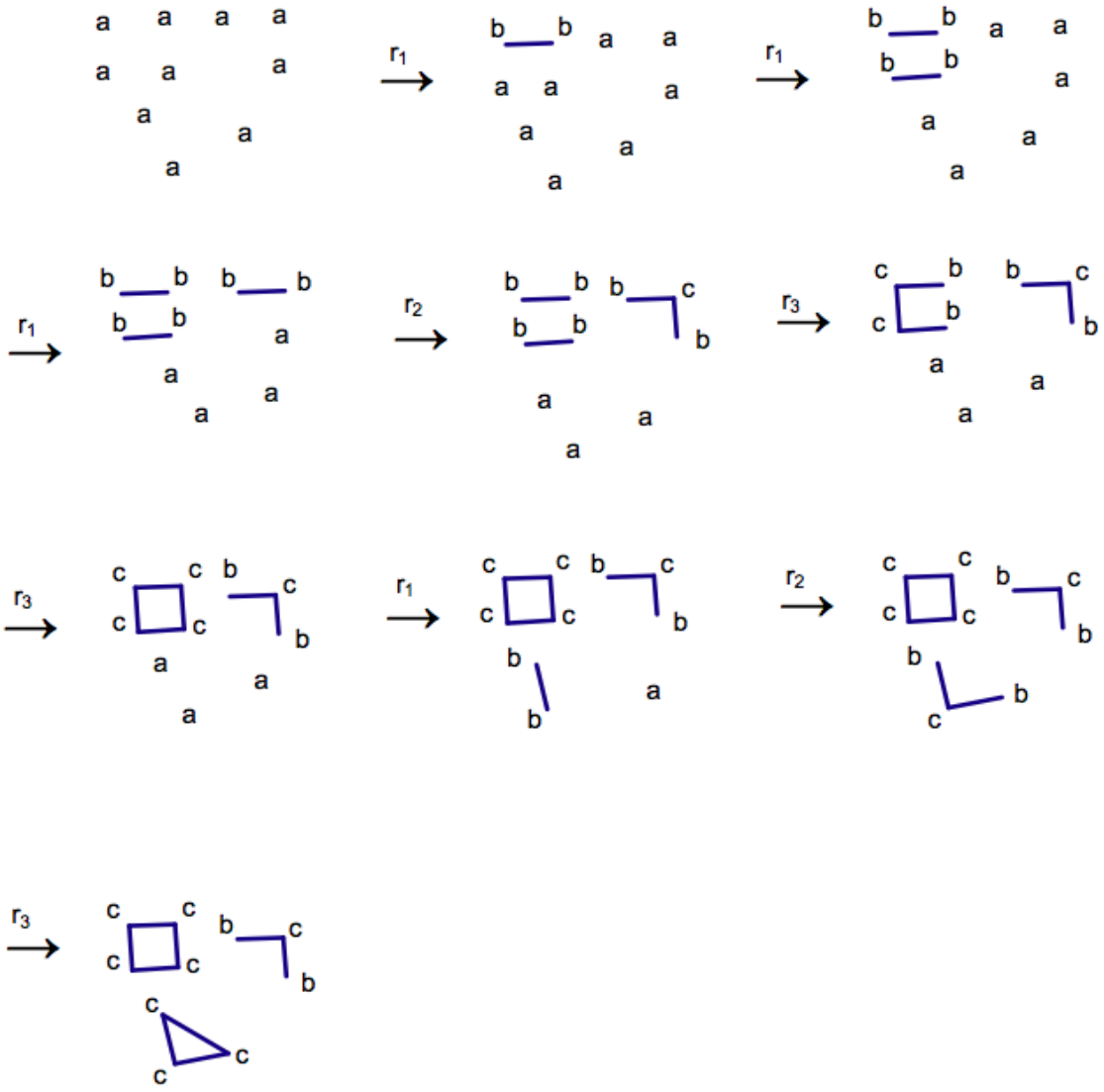


Figure 19: An example trajectory or assembly sequence

edges are empty sets. Now that we have applied rule r_1, r_2 also become applicable since we have vertices labeled b in the graph G_1. Similarly, we can also apply r_3 once we apply r_2. An example trajectory of this assembly process is shown in Figure 19. Note that since the system is nondeterministic it can have more than one trajectory or assembly sequence. We can see in Figure 19 that the reachable set $\mathcal{R}(G_0, \Phi)$ and also the set of reachable components $\mathcal{C}(G_0, \Phi)$, as defined in Definition 15 and 16 respectively, consist of only cycles and chains. But the cycles are the stable components since there is no rule in which the left hand side graph vertices are labeled by c, i.e. there is no such label-preserving embedding or witness

which is needed to apply a rule . On the other hand, we can still apply rule r_3 to the chains since we have two vertices labeled b in graph G_8 in the trajectory shown in Figure 19. Thus the stable set $S(G_0, \Phi)$, as defined in Definition 17, consists of cycles only.

Let us consider another rule set $\Phi^{'}$ as defined by

$$\Phi^{'} = \begin{cases} a \quad a \Rightarrow e - q, & (r_1) \\ a \quad e \Rightarrow d - u, & (r_2) \\ a \quad d \Rightarrow c - t, & (r_3) \\ a \quad c \Rightarrow b - s, & (r_4) \\ a \quad b \Rightarrow p - r, & (r_5) \\ \begin{array}{ccc} & u & \\ q & & t \end{array} \Rightarrow \begin{array}{ccc} & u & \\ q^{'} & \text{——} & t \end{array}, & (r_6) \\ \begin{array}{ccc} & t & \\ q^{'} & & s \end{array} \Rightarrow \begin{array}{ccc} & t & \\ q^{''} & \text{——} & s \end{array}, & (r_7) \\ \begin{array}{ccc} & s & \\ q^{''} & & r \end{array} \Rightarrow \begin{array}{ccc} & s & \\ q^{'''} & \text{——} & r \end{array}, & (r_8) \\ \begin{array}{ccc} & r & \\ q^{'''} & & p \end{array} \Rightarrow \begin{array}{ccc} & r & \\ q^{''''} & \text{——} & p \end{array}, & (r_9) \\ q^{''''} - t \Rightarrow q^{''''} \quad t, & (r_{10}) \\ q^{''''} - s \Rightarrow q^{''''} \quad s, & (r_{11}) \\ q^{''''} - r \Rightarrow q^{''''} \quad r, & (r_{12}) \end{cases} \tag{14}$$

Note that the rule set $\Phi^{'}$ consists of binary and ternary rules. It also contains *destructive rules* r_{10}, r_{11} and r_{12}. An example trajectory is shown in Figure 20. Here the intial graph G_0 is same as the one we used for the rule set Φ.

We can see in Figure 20 that the reachable set $\mathcal{R}(G_0, \Phi)$ and also the set of reachable components $\mathcal{C}(G_0, \Phi)$ consist of cycles and chains. But the cycles are the only stable component. Also note that there is only one cycle of length six

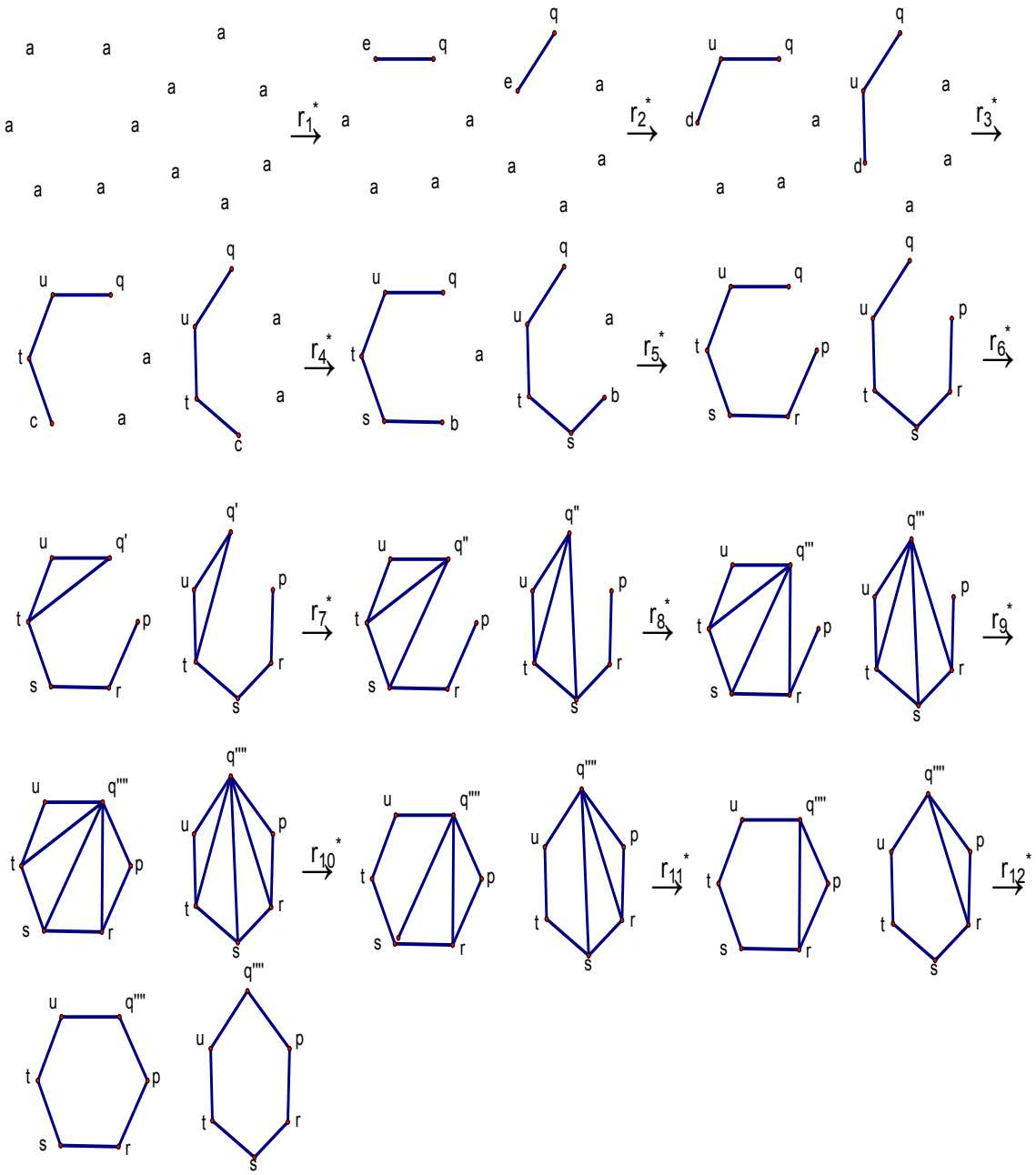


Figure 20: An example trajectory of the system (G_0, Φ')

unlike the previous example where we have cycles of various length. Thus rule set Φ' produces an unique stable component.

3.2. Topological Properties of the Graph Grammar Model

Now that we are familiar with some basic terminologies of graph grammar and how these grammars generate stable components, we should consider the problem

of constructing *uniquely stable components*. In the previous example of assembly system (G_0, Φ) all cycles of length three or greater are stable. But in the assembly system (G_0, Φ') only cycles of length six are stable. The question is "how can we construct a cycle or cycles of exactly one particular length?" that is, an unique stable component. According to Klavins et al.,

> it is intuitively clear that a grammar containing only binary rules can not distinguish between a cycle of length $2N$, and two identical cycles of length N[4]

. Klavins et al. suggests that a rule set consisting of larger rules can construct a stable cycle of particular size. Let us consider another situation as illustrated by Klavins et al. [4, 5]. We'll take our rule set Φ_2 to be the one shown below:

$$\Phi_2 = \begin{cases} a \quad a \Rightarrow b - c, & (r_1) \\ a \quad c \Rightarrow e - d, & (r_2) \\ a \quad e \Rightarrow g - f, & (r_3) \\ \begin{matrix} & d & \\ b & & f \end{matrix} \Rightarrow \begin{matrix} & d_1 & \\ b_1 & - & f_1 \end{matrix}, & (r_4) \\ \begin{matrix} & f_1 & \\ b_1 & & g \end{matrix} \Rightarrow \begin{matrix} & f_2 & \\ b_2 & - & g_1 \end{matrix}, & (r_5) \\ b_2 - f_2 \Rightarrow b_3 \quad f_3, & (r_6) \end{cases} \tag{15}$$

Note that our rule set Φ_2 consists of two ternary rules r_4 and r_5, and also one destructive rule r_6. Now suppose we have an initial graph G_0 and an example trajectory or assembly sequence as depicted in Figure 21. Here the stable set consists of an unique cycle of length four. If instead of the ternary rules, we used a binary rule of the form $b \quad g \Rightarrow b_1 - g_1$, then stable set would contain cycles of length 4, 8, 12 and so on. This limitation is implied by Theorem 5 discussed below.

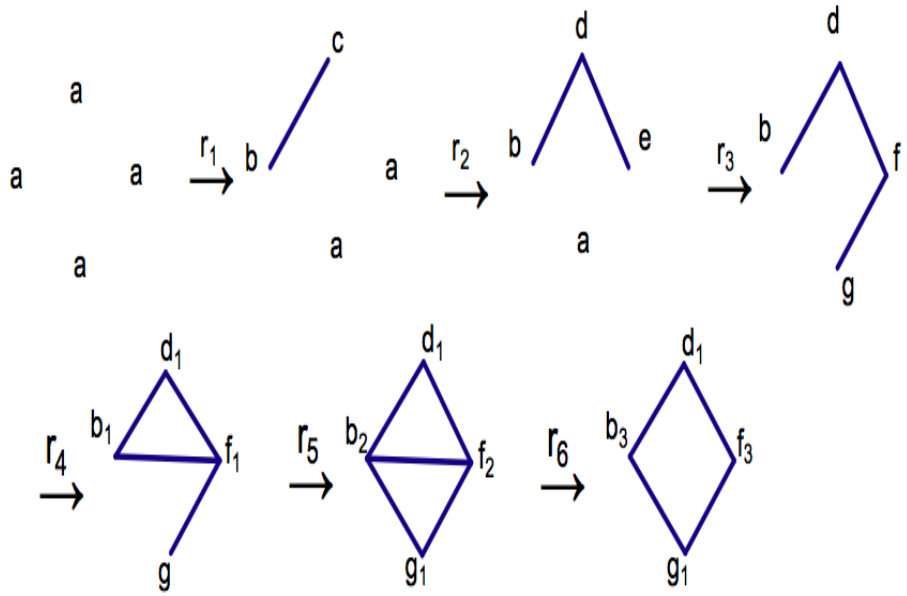


Figure 21: An example trajectory or assembly sequence for the assembly system (G_0, Φ_2). Source:[5]

Klavins et al. bounded the size of reachable and stable sets of an assembly system (G_0, Φ) using an algebraic topological tool, called *covering space theory*. First we give a formal definition of a *cover* of a graph.

Definition 18. Given a graph G, an *n-fold cover* of G is a graph $\tilde{G}$ such that there exists a label-preserving *n-to-1* continuous map $p : \tilde{G} \to G$ which is a local homeomorphism.

Theorem 5. *Let (G_0, Φ) is an assembly system with an acyclic rule set Φ. Then the set of reachable components $\mathcal{C}(G_0, \Phi)$ is closed under covers. In particular, $\mathcal{C}(G_0, \Phi)$ contains infinitely many isomorphism types of graphs if it contains any graph with a cycle.*

A proof of this theorem can be found in Klavins et al. papers [4, 5]. We illustrate the same proof with an example.

Let us consider a part of an assembly sequence or trajectory of an assembly system (G_0, Φ) where Φ is the rule set as shown in subsection 3.1. Part of this assembly sequence or trajectory α is shown in Figure 22 .

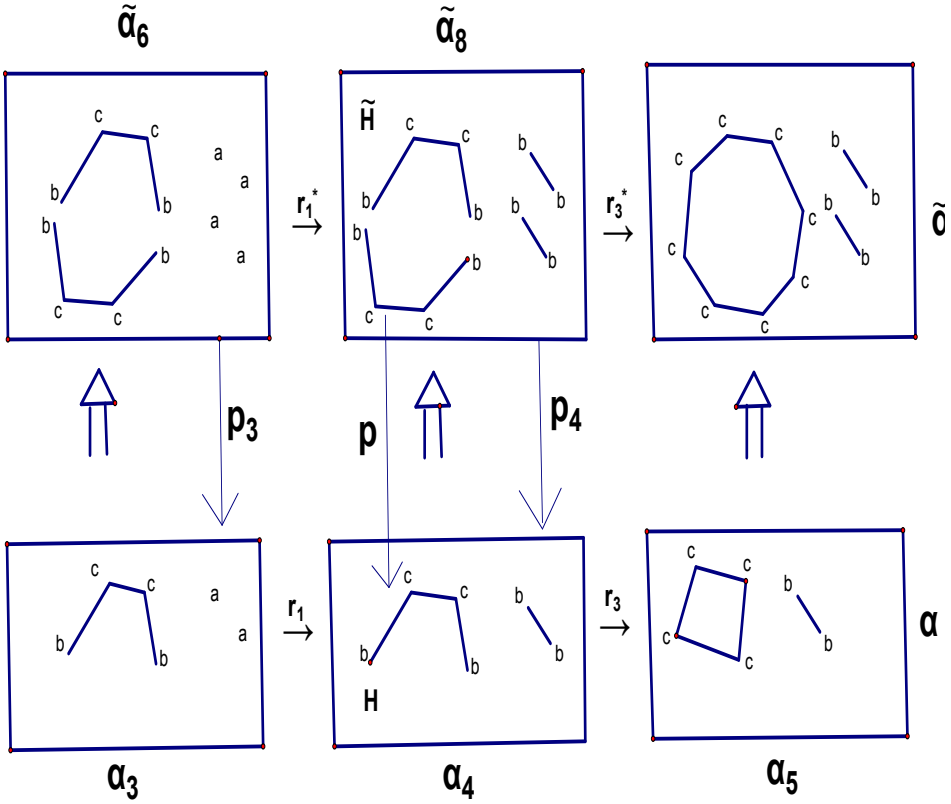


Figure 22: Part of a trajectory or assembly sequence for the assembly system (G_0, Φ) and its lifting.

We pick $H = \begin{matrix} c - c \\ | \quad | \\ b \quad b \end{matrix} \in \mathcal{C}(G_0, \Phi)$ as a component of 4^{th} graph in the trajectory α, that is, α_4. Suppose $\tilde{H}$ is a 2-fold cover of H with a covering projection $p : \tilde{H} \to H$ as shown in Figure 22. We are going to reverse or disassemble the trajectory σ and then lift the disassembly to build another trajectory $\tilde{\alpha}$ with $\tilde{H}$, a component of $\tilde{\alpha_8}$. Here $\tilde{\alpha_8}$ is the graph consisting of disjoint union of $\tilde{H}$ with 2 disjoint copies of the complement $\alpha_4 - H$. We define the projection $p_4 : \tilde{\alpha_8} \to \alpha_4$ via p on $\tilde{H}$ and the projection of the copies of $\alpha_4 - H$.

We can see that the image of the right hand side of the rule r_1 (i.e. the rule which produces α_4), $h_4(R_1)$ is a subtree of α_4. Now we have $(\tilde{\alpha_8}, p_4)$ is a covering space of α_4, R_1 is a connected and locally arcwise-connected space, and $b \in R_1$, $b \in \tilde{\alpha_8}$, and $b = p_4(b)$ and we have a map $h_4 : (R_1, b) \to (\alpha_4, b)$. Since R_1 is acyclic the fundamental group $\pi(R_1, b)$ contains only the identity element and the subgroup $h_{4*}\pi(R_1, b)$ is a subset of $p_{4*}\pi(\tilde{\alpha_8}, b)$. Now according to the lifting

property of arbitrary maps to a covering space (Theorem 5.1 in [8]), it follows that there exists a lifting $\tilde{h_4} : (R_1, b) \to (\tilde{\alpha_8}, b)$ and the inverse image $\tilde{R_1} = {p_4}^{-1}(R_1)$ is a disjoint union of isomorphic copies of R_1.

Now we can reverse the assembly by replacing each R_1 in $\tilde{R_1}$ with the left hand side of the rule L_1 2 times since we have 2-fold cover. Suppose after 2 replacements we obtain a graph $\tilde{\alpha}_6$ and $\tilde{L_1}$ is the disjoint union of 2 copies of L_1 within the graph $\tilde{\alpha}_6$. Now we define a projection map $p_3 : \tilde{\alpha}_6 \to \alpha_3$, where α_3 is the 3^{rd} graph in the trajectory α, to be (i) p_4 on the complement of $\tilde{L_1}$; and (ii) the projection $\tilde{L_1} \to L_1$ identifying the disjoint copies. Now the graph $\tilde{\alpha_6}$ is obviously a 2-fold cover of α_3 via p_3 since labels and indices are preserved. Now if we continue this process inductively, it will eventually terminate in a covering projection $p_0 : \tilde{\alpha_0} \to \alpha_0$, where α_0 is the initial graph in the trajectory α and $\alpha_0 = G_0$. Since we know that the lift of any discrete set is a discrete set, we have that $\tilde{\alpha_0}$ is isomorphic to G_0. Thus $\tilde{\alpha} \in \tau(G_0, \Phi)$ and $H \in \mathcal{C}(G_0, \Phi)$.

3.3. Algorithms for Generating Graph Grammar

Klavins et al. considered a question: Given a graph G and an initial graph G_0, can we find a set of rules Φ such that $S(G_0, \Phi) = \{G\}$(up to isomorphism and not considering labels)? They devised an algorithm [4, 5] for constructing a rule set or grammar to assemble a given tree (i.e. an acyclic graph) and then used it to build another algorithm [4, 5] for constructing a rule set to assemble any arbitrary graph.

3.3.1. Algorithm for Generating Graph Grammar for Any Tree

Algorithm 2 defines a recursive procedure *CreateTree* that, given any tree T, produces a set of binary rules Φ_T such that $S(G_0, \Phi_T) = \{T\}$ [4, 5].

The algorithm takes an unlabeled tree (V, E) as input and returns (Φ, l) where

Φ is the rule set and l is a labeling function on V. In lines 1-2 the singleton graph is labeled with the label a. Next an edge $\{p, q\}$ is chosen randomly from the set of edges E in line 4. Now in lines 5-6 *CreateTree* is called recursively on the two tree components (V_1, E_1) and (V_2, E_2) resulting from deletion of $\{p, q\}$ from E. The recursive calls on (V_1, E_1) and (V_2, E_2) return two rule sets Φ_1 and Φ_2 respectively and also two labeling functions l_1 and l_2 respectively in line 7. The algorithm uses new labels m and n to label the vertices of the right hand side graph of the new rule. Now in line 9 the final rule set Φ is formed by combining the two rule sets Φ_1 and Φ_2. The new labeling function l is constructed in line 10 from the two previously derived labeling functions l_1 and l_2.

Algorithm 2 *CreateTree*(V, E)

Require: $T = (V, E)$ is an unlabeled tree

1: **if** $V = \{p\}$ **then**
2: **return** $(\emptyset, \{(p, a)\})$
3: **else**
4: Choose any edge $\{p, q\} \in E$
5: let (V_1, E_1) be the component of $(V, E - \{p, q\})$ containing p.
6: let (V_2, E_2) be the component of $(V, E - \{p, q\})$ containing q.
7: let $(\Phi_k, l_k) = \mathit{CreateTree}(V_k, E_k)$ for $k = 1, 2$
8: let m, n be new labels
9: $\Phi = \Phi_1 \cup \Phi_2 \cup \{l_1(p) \quad l_2(q) \Rightarrow m - n\}$
10: $l = (l_1 - \{(p, l_1(p))\}) \cup (l_2 - \{(q, l_1(q))\}) \cup \{(p, m), (q, n)\}$
11: **return** (Φ, l)
12: **end if**

In Figure 23 we give an example of synthesizing a tree using the *CreateTree* procedure described above. Here we have a tree $T = (V, E)$ where $V = \{1, 2, 3, 4, 5, 6\}$ and $E = \{\{1, 2\}, \{2, 3\}, \{2, 4\}, \{4, 5\}, \{4, 6\}\}$.

Since there is no singleton or isolated vertex in T, we can skip base cases(lines $1 - 2$) for the moment. Now according to the lines $3 - 4$, we pick an arbitrary edge $\{2, 4\} \in E$, and remove it from the tree T, which results into two separate tree components (V_1, E_1) and (V_2, E_2) where $V_1 = \{1, 2, 3\}$, $V_2 = \{4, 5, 6\}$ and

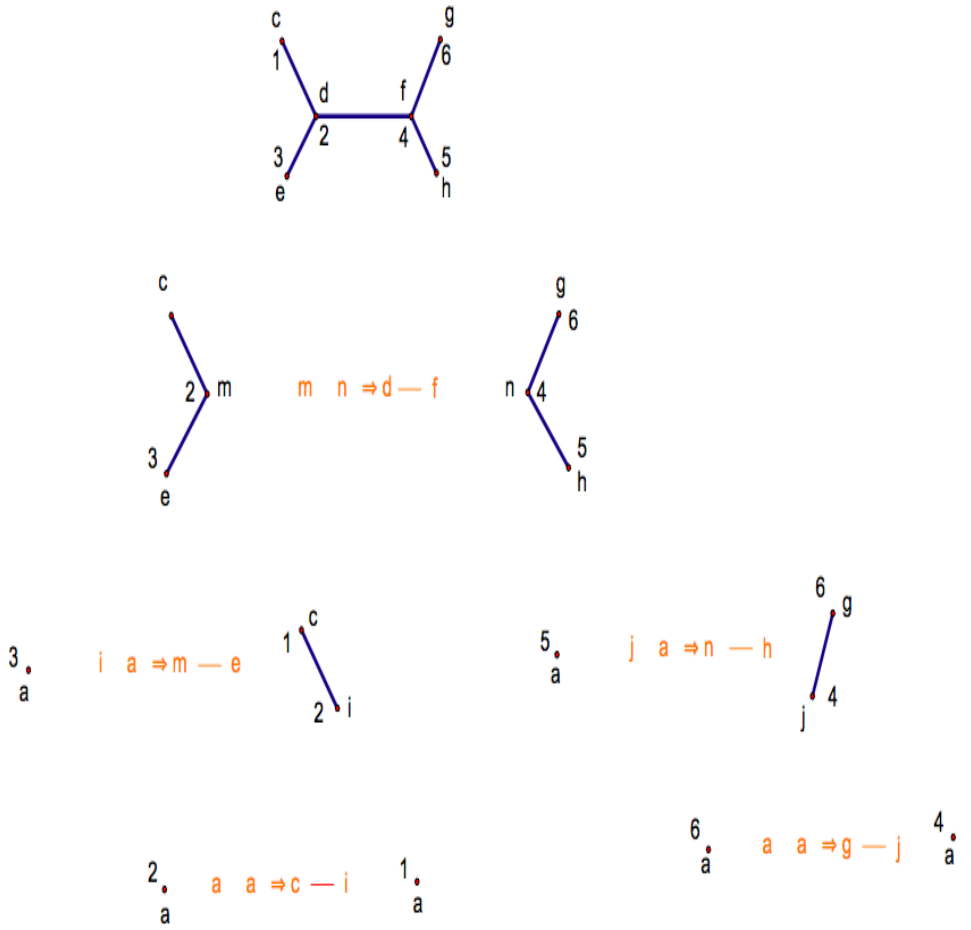


Figure 23: Generating rule set or grammar to produce a tree using *CreateTree* procedure

$E_1 = \{\{1,2\},\{2,3\}\}$ and $E_2 = \{\{4,5\},\{4,6\}\}$. Now line 7 makes a recursive call to the *CreateTree* procedure and as we do not have any single vertex this time, the procedure goes to line $8-9$, according to which a rule $m \quad n \Rightarrow d - f$ is to be added to the rule set Φ, and the two vertices 2 and 4 have two new labels m and n, respectively. Then *CreateTree* procedure is called recursively on the two components (V_1, E_1) and (V_2, E_2), which generates the rule sets Φ_1 and Φ_2 respectively. In the end the binary rule set or grammar Φ looks like the following.

$$\Phi = \begin{cases} a \quad a \Rightarrow g - j, & (r_1) \\ a \quad a \Rightarrow c - i, & (r_2) \\ i \quad a \Rightarrow m - e, & (r_3) \\ j \quad a \Rightarrow n - h, & (r_4) \\ m \quad n \Rightarrow d - f, & (r_5) \end{cases} \tag{16}$$

Also note that in the end all of the vertices get labeled a, which gives us the intial graph G_0, as depicted in Figure 23.

3.3.2. Algorithm for Generating Graph Grammar for Any Arbitrary Graph

Given a graph G, Algorithm 3 defines a function *CreateGraph* that produces a rule set Φ_G such that $S(\Phi_G) = \{G\}$ [4, 5]. Note that an aribtrary graph may have cycle in it. Since according to Theorem 5 binary rules can not produce an uniquely stable cycle, we need a rule set Φ_G consisting of rules of larger sizes to produce an uniquely stable graph.

The function *CreateGraph* takes an unlabeled graph (V, E) as input and returns (Φ, l), where Φ is a rule set and l is a labeling function on V. First we find a *maximal spanning tree* $T = (V, E_T)$ by using an algorithm called *Kruskal's Algorithm* [16] in line 1. Now since we already have Algorithm 2 to generate rule set for any tree, we call the procedure *CreateTree* on $T = (V, E_T)$ to construct a rule set Φ and a new labeling function l for this maximal spanning tree T in line 2. We assign the label of the last vertex visited while executing *CreateTree* to a in line 4. Next in line 6-14, we pick every edge $\{v, v'\}$ that is not in the edge set E_T of the maximal spanning tree T, that is, the additional edges creating cycles in the original graph G and then the *Triangulate* function is called in line 7-8. The *Triangulate* function finds the shortest path between r and v and if the length of

the shortest path is greater than one then a rule of the form Figure 25(b) is added to rule set for each vertex along the shortest path and also new edges are added to the spanning tree. Next another rule of the form as given in Figure 25(a) is added to the rule set Φ and an edge $\{v, v^{'}\}$ is also added closing the spanning tree. We remove all the extra edges between r and v in E_S by generating destructive rules in the same order we generated the triangular rules. Note that we change the label of r or root node at every step of triangulation to maintain the order of the rules. This rule set Φ produces a graph isomorphic to G and the only element in the stable set for Φ, that is, the uniquely stable element.

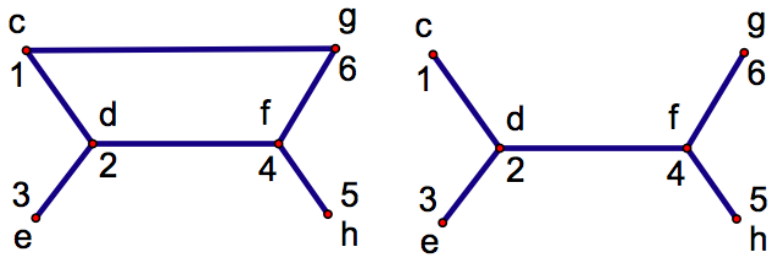


Figure 24: (a) An arbitrary graph G; (b) Maximal spanning tree T of G

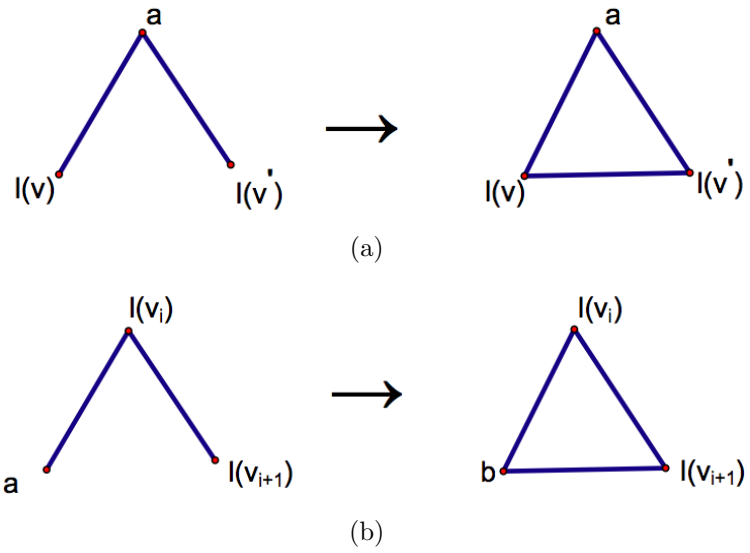


Figure 25: Ternary rules used in Algorithm 3. Source:[4]

We illustrate Algorithm 3 with an example of synthesizing an arbitrary graph G using the *CreateGraph* procedure. First of all, we need a graph G, same as the one we used in the previous example of Algorithm 2 except the fact that the set of

Algorithm 3 *CreateGraph*(V, E)

Require: $G = (V, E)$ is an unlabeled connected graph.

let $T = (V, E_T)$ be a maximal spanning tree of (V, E)
let $(\Phi, l) =$ *CreateTree*(V, E_T)
let r be the last vertex visited while executing $CreateTree$.
let $a = l(r)$.
let $E_S = E_T$
for all $e = \{v, v^{'}\} \in E - E_T$ **do**
    let $(\Psi, E_S, a) =$ *Triangulate*(V, E_S, r, v, l, a)
    let $(\Psi^{'}, E_S, a) =$ *Triangulate*$(V, E_S, r, v^{'}, l, a)$
    let b be a previously unused label
    let ψ be the rule rule denoted in Figure 4(a)
    let $\Phi = \Phi \cup \Psi \cup \Psi^{'} \cup \psi$
    $E_S = E_S \cup \{\{v, v^{'}\}\}$
    let $a = b$
end for
for all $v \in V$ **do**
    if $\{r, v\} \in E_S - E$ **then**
        let ψ be the rule $a - l(v) \Rightarrow a \quad l(v)$
        let $\Phi = \Phi \cup \psi$
    end if
end for
return Φ

FUNCTION DEFINITION: $Triangulate(V, E, r, v, l, a)$

let $\Psi = \emptyset$
let $(v_1, \ldots, v_n)$ be the shortest path in G from r to v
for $k = 2$ to $n - 1$ **do**
    let b be a previously unused label
    let ψ be the rule denoted in Figure 4(b)
    let $\Psi = \Psi \cup \{\psi\}$
    let $E = E \cup \{\{r, v_{i+1}\}\}$
    let $a = b$
end for
return (Ψ, E, a)

edges E has one more edge $\{1,6\}$ in it, as shown in Figure 24(a). Now according to Algorithm 3, we find a maximal spanning tree of G using *Kruskal's algorithm*[16]. Figure 24(b) shows the maximal spanning tree $T = (V, E_T)$ derived from the graph G. Note that, the maximal spanning tree T is similar to the tree that we used in the previous example of Algorithm 2 in Figure 23. Thus by line 2 in Algorithm 3 , that is after applying *CreateTree* procedure on T, we have the similar rule set Φ as shown in equation (11). By line 3-4, $a = l(r) = f$ where $r = 4$ is the last vertex visited while executing *CreateTree* procedure. Line 5 assigns the edges of T, E_T, to E_S, that is, $E_S = \{\{1,2\},\{2,3\},\{2,4\},\{4,5\},\{4,6\}\}$. We see that $\{v, v^{'}\} = \{1,6\}$ is the only edge in E, but not in E_T, that is, $\{1,6\} \in E - E_T$. Thus by line 6-7 function $Triangulate(V, E_S, 4, 1, l, f)$ is called. Now by line 2 in the $Triangulate$ definition , the shortest path from 4 to 1 is $4 - 2 - 1$ which is of length 2. According to the *for* loop in line 3 of $Triangulate$ definition and the Figure 25(b) a ternary rule r_6 is added to the rule set Ψ, an edge $\{4,1\}$ is also added to E_S (as shown in Figure 26(a)), and $a = f^{'}$.

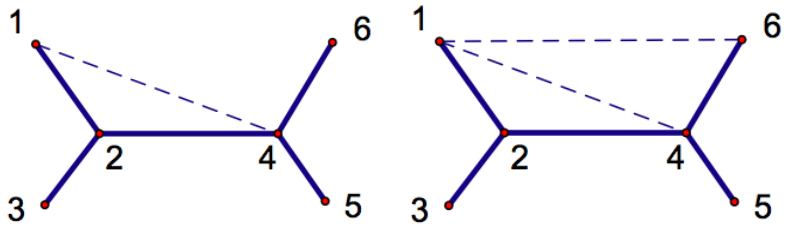


Figure 26: Addition of edges to the assembly graph to produce ternary rules as described in Algorithm 3

Then in line 8 in the *CreateGraph* procedure function $Triangulate(V, E_S, 4, 6, l, f^{'})$ is called, going back to the *Triangulate* definition. But since the shortest path from 4 to 6 is of length 1, $Triangulate(V, E_S, 4, 6, l, f^{'})$ returns $(\Psi^{'}, E_S, f^{'})$ where $\Psi^{'} = \Psi$. Now by line 10 of the *CreateGraph* procedure and also by Figure 25(a) another ternary rule r_7 and Ψ are added to the rule set or grammar Φ. In line 12, an edge $\{1,6\}$ is added to the E_S (as shown in Figure 26(b)). Line 13 assigns $f^{''}$ to a. In line 15-18 , a destructive rule r_8 of the form $f^{''} - c \Rightarrow f^{''} \quad c$ is added to

rule set Φ to delete the extra edge $\{4, 1\}$ in E_S. Line 21 returns the final rule set Φ as shown below.

$$\Phi = \begin{cases} a \quad a \Rightarrow g - j, & (r_1) \\ a \quad a \Rightarrow c - i, & (r_2) \\ i \quad a \Rightarrow m - e, & (r_3) \\ j \quad a \Rightarrow n - h, & (r_4) \\ m \quad n \Rightarrow d - f, & (r_5) \\ \begin{array}{ccc} & d & \\ f & \diagdown & c \end{array} \Rightarrow \begin{array}{ccc} & d & \\ f' & \diagup\!\diagdown & c \end{array}, & (r_6) \\ \begin{array}{ccc} & f' & \\ c & \diagdown & g \end{array} \Rightarrow \begin{array}{ccc} & f'' & \\ c & \diagup\!\diagdown & g \end{array}, & (r_7) \\ f'' - c \Rightarrow f'' \quad c, & (r_8) \end{cases} \tag{17}$$

Figure 27 illustrates an assembly of G as an uniquely stable component using the rule set or grammar Φ to verify the correctness of Algorithm 3.

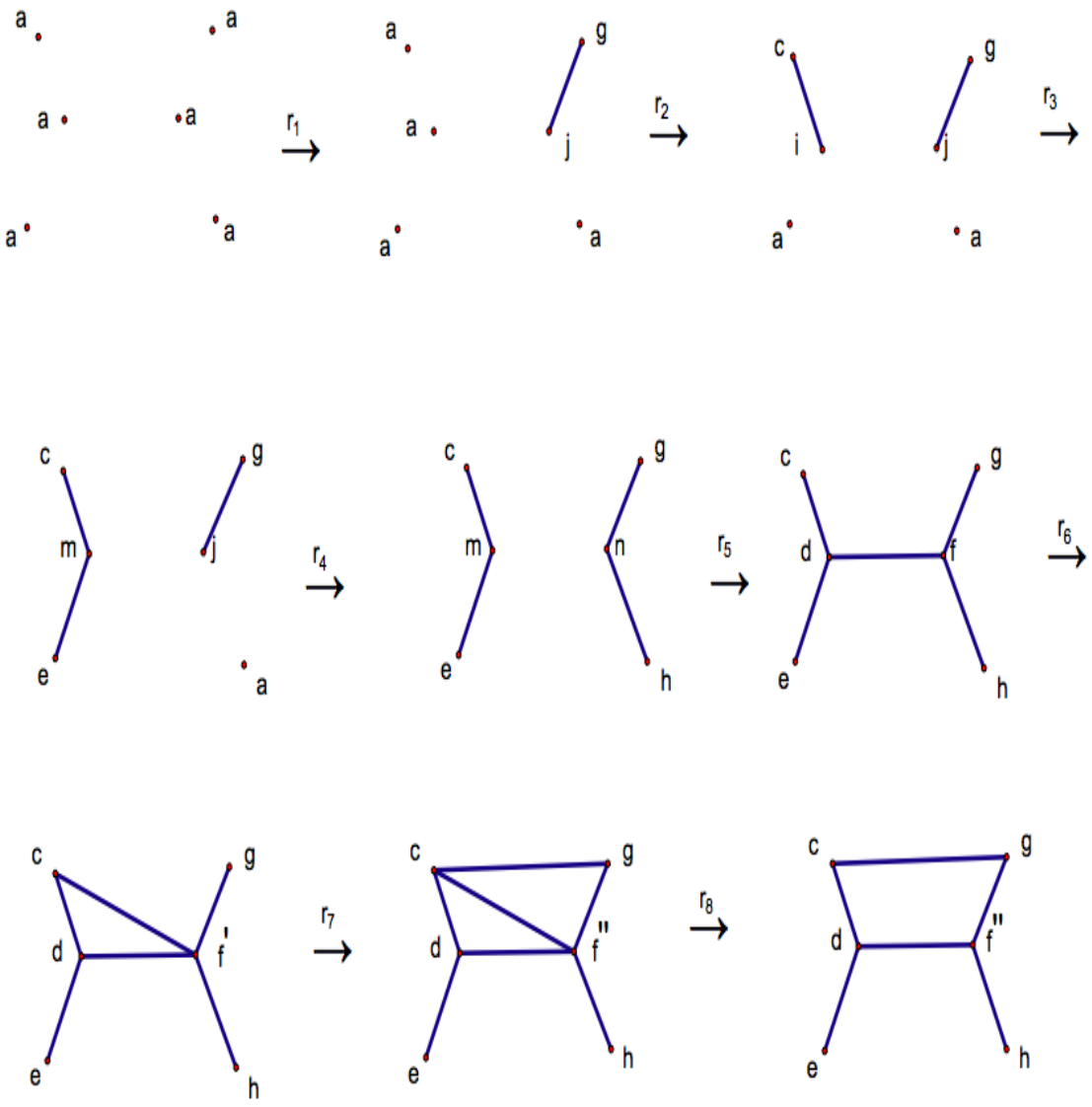


Figure 27: An assembly of the graph G using the rule set Φ produced by *CreateGraph*

References

[1] J. Bishop, S. Burden, E. Klavins, R. Kreisberg, W. Malone, N. Napp and T. Nguyen, *Programmable Parts: A Demonstration of the Grammatical Approach to Self-Organization*, International Conference on Intelligent Robots and Systems, (2005).

[2] P. J. G. Butler, *Assembly of Tobacco Mosaic Virus*, Phil. Trans. R. Soc. Lond. B (1976) 276, pp. 151-163.

[3] P. J. G. Butler, *Self-Assembly of Tobacco Mosaic Virus: The Role of An Intermediate Aggregate in Generating Both Specificity and Speed*, Phil. Trans. R. Soc. Lond. B (1999) 354, pp. 537-550.

[4] R. Ghrist, E. Klavins, D. Lipsky, *Graph Grammars for Self Assembling Robotic Systems*, Proc. Int. Conf. Robotics and Automation, (2004).

[5] R. Ghrist, E. Klavins, and D. Lipsky, *A Grammatical Approach to Self-Organizing Robotic Systems*, IEEE Trans. Automatic Controls, 51(6), 949-962, (2006).

[6] M. Jackiela and K. Saitou, *Automated Optimal Design of Mechanical Conformational Switches*, Artificial Life, 2 (1995), pp. 129-156.

[7] M. Jackiela and K. Saitou, *On Classes of One-Dimensional Self-Assembling Automata*, Complex Systems, 10 (1996), pp. 391-416.

[8] W. S. Massey, *A Basic Course in Algebraic Topology*, Springer-Verlag, (1991).

[9] N. Papadakis, P. W. K. Rothemund, E. Winfree, *Algorithmic Self-Assembly of DNA Sierpinski Triangles*, PLoS Biology, 2 (2004), pp. 2041-2053.

[10] J. A. Pelesko, *Self Assembly The Science of Things That Put Themselves Together*, Chapman & Hall/CRC Press, (2007).

[11] P. W. K. Rothemund and E. Winfree, *The Program-Size Complexity of Self-*

Assembled Squares, Symposium on Theory of Computing (STOC 2000), Portland, OR, (2000).

[12] K. Saitou, *Conformational Switching in Self-Assembling Mechanical Systems*, IEEE Transactions on Robotics and Automation, 15 (1999), pp. 510–520.

[13] E. Winfree, *Algorithmic Self-Assembly of DNA*, Ph.D. Thesis, California Institute of Technology, (1998).

[14] *http://homepages.cae.wisc.edu/ stone/bubble%20raft%20movies.htm*

[15] *http://en.wikipedia.org/wiki/File:Halit-Kristalle.jpg*

[16] *http://en.wikipedia.org/wiki/Kruskal's_algorithm*

[17] *http://en.wikipedia.org/wiki/File:TobaccoMosaicVirus.jpg*